

2

NAVAL POSTGRADUATE SCHOOL Monterey, California

AD-A232 555



DTIC
ELECTE
MAR 6 1991
S B D

THESIS

SOFTWARE TESTING
FOR
EVOLUTIONARY ITERATIVE RAPID PROTOTYPING

by

Edward V. Davis, Jr.

December, 1990

Thesis Advisor:

Timothy J. Shimeall

Approved for public release; distribution is unlimited.

91 3 04 002

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) SOFTWARE TESTING FOR EVOLUTIONARY ITERATIVE RAPID PROTOTYPING(U)			
12. PERSONAL AUTHOR(S) Davis, Edward V., Jr.			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) December 1990	15. PAGE COUNT 295
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Rapid prototyping is emerging as a promising software development paradigm. It provides a systematic and automatable means of developing a software system under circumstances where initial requirements are not well known or where requirements change frequently during development. To provide high software quality assurance requires sufficient software testing. The unique nature of evolutionary iterative prototyping is not well-suited for classical testing methodologies, therefore the need exists for a testing methodology tailored for this prototyping paradigm. This thesis surveys current prototyping and testing practices to provide a foundation for developing a software testing methodology for prototyping. The thesis then describes a testing methodology for rapid prototyping, Spiral Testing, and the Test Goal Tracking System (TGTS), a requirements-based testing tool developed for use with the Computer Aided Prototyping System (CAPS). TGTS provides the first in an anticipated family of testing tools to support the CAPS environment. This thesis shows key prototyping characteristics impinging on testing, the value of Spiral Testing and the feasibility and qualities of complementary testing tools to support evolutionary iterative rapid prototyping.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Timothy Shimeall		22b. TELEPHONE (Include Area Code) (408) 646-2509	22c. OFFICE SYMBOL CS/Sm

Approved for public release; distribution is unlimited.

Software Testing
for
Evolutionary Iterative Rapid Prototyping

by

Edward V. Davis, Jr.
Major, United States Marine Corps
B.S., United States Naval Academy

Submitted in partial fulfillment
of the requirements for the degree of

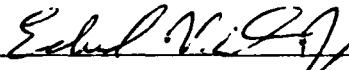
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL


December 1990

Author:

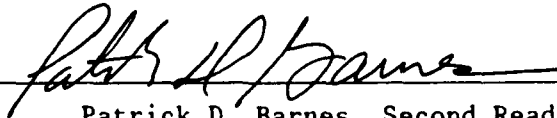


Edward V. Davis, Jr.

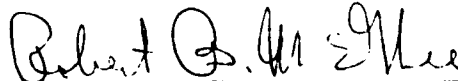
Approved by:



Timothy J. Shimeall, Thesis Advisor



Patrick D. Barnes, Second Reader



Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

Rapid prototyping is emerging as a promising software development paradigm. It provides a systematic and automatable means of developing a software system under circumstances where initial requirements are not well known or where requirements change frequently during development. To provide high software quality assurance requires sufficient software testing. The unique nature of evolutionary iterative prototyping is not well-suited for classical testing methodologies, therefore the need exists for a testing methodology tailored for this prototyping paradigm.

This thesis surveys current prototyping and testing practices to provide a foundation for developing a software testing methodology for prototyping. The thesis then describes a testing methodology for rapid prototyping, Spiral Testing, and the Test Goal Tracking System (TGTS), a requirements-based testing tool developed for use with the Computer Aided Prototyping System (CAPS). TGTS provides the first in an anticipated family of testing tools to support the CAPS environment. This thesis shows key prototyping characteristics impinging on testing, the value of Spiral Testing and the feasibility and qualities of complementary testing tools to support evolutionary iterative rapid prototyping.



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

THESIS DISCLAIMER

DBase and dBase III+ are registered trademarks of Ashton-Tate.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. SIGNIFICANCE OF RAPID PROTOTYPING	1
1. The Need for Prototyping	2
a. The Software Gap	2
b. Software System Reliability and Maintainability	2
2. Purpose of Prototyping	3
a. Prototyping as a Concept	3
b. Prototyping Contrasted with the Waterfall Life Cycle Model	6
3. Techniques of Rapid Prototyping and Testing Implications	9
4. Strengths and Weaknesses of Prototyping	10
5. Prototyping Today	11
B. SIGNIFICANCE OF SOFTWARE TESTING	12
1. Purpose of testing	12
2. Conventional testing and its limitations in prototyping	14
C. PURPOSE OF THESIS	17
D. SCOPE OF THESIS	19
1. Need for a Companion Testing Methodology in Rapid Prototyping	20

2. Overview of Thesis	22
II. SURVEY OF PRECEDING WORK	24
A. SELECTIVE OVERVIEW OF RAPID PROTOTYPING	24
1. The Promise and Value of Prototyping	24
2. Analysis of existing prototyping methodologies	27
a. Rapid Throwaway Prototype Methodology	28
b. Incremental Development Methodology	29
c. Evolutionary Prototyping Methodology	30
d. Reusable software components methodology	31
e. Automated Software Synthesis Methodology	32
3. Conclusions on prototyping methodologies	34
B. SOFTWARE TESTING: PURPOSE AND SELECTED AUTOMATED METHODOLOGIES	34
1. The Software Testing Process	36
2. Software Testing Methodologies and Companion Testing Tools	37
a. Requirements-based testing	39
b. Functional Testing	43
c. Structural Testing	44
3. A Brief Review of Selected Automated Testing Tools	49
(1) ASSET - A System to Select and Evaluate Tests.	49
(2) UNISEX: a UNIX-based Symbolic EXecutor for Pascal.	50

C.	SYNOPSIS OF SOFTWARE TESTING METHODS WITHIN RAPID PROTOTYPING	51
D.	BOEHM'S SPIRAL MODEL OF SOFTWARE DEVELOPMENT	53
E.	THE CAPS RAPID-PROTOTYPING SYSTEM	55
1.	CAPS	56
2.	PSDL	58
III.	A PROPOSED TESTING METHODOLOGY AND TOOL DEVELOPMENT STRATEGY FOR RAPID PROTOTYPING	62
A.	RAPID PROTOTYPING SYSTEM CHARACTERISTICS	62
B.	PROTOTYPE-BASED TESTING	62
1.	Test Information From Iteration	63
2.	Test Information About Components	64
3.	Test Information About Performance	65
4.	Recording Test Information	66
C.	OVERVIEW OF A TESTING SUPPORT SYSTEM FOR EVOLUTIONARY, ITERATIVE RAPID PROTOTYPING	66
1.	Shimeall's Testing Within Iterative Rapid Prototyping	66
2.	Testing-related Features Needed in a Prototyping System	70
a.	Requirements-capturing Features	70
b.	Reusable Components Features	70
c.	Prototyping Language Features	71
3.	Spiral Testing: A Testing Methodology to Support Evolutionary Iterative Rapid Prototyping	72
a.	The First Test Planning Iterations	73

b.	Subsequent Test Planning Iterations	75
c.	The Final Test Planning Iterations	80
d.	Spiral Testing: Advantages and Disadvantages . .	81
e.	Characteristics of Candidate Software Testing Tools	82
D.	REQUIREMENTS-BASED TESTING: A KEY METHODOLOGY FOR RAPID PROTOTYPING	83
IV.	TGTS: A SAMPLE REQUIREMENTS-BASED TESTING TOOL	87
A.	AN OVERVIEW OF THE TEST GOAL TRACKING SYSTEM	87
1.	TGTS Distinction	87
2.	TGTS Goals	88
3.	TGTS Database Software	89
B.	DEVELOPMENT AND DESIGN OF THE TEST GOAL TRACKING SYSTEM .	90
1.	Detailed Design Decisions and Tool Structure for TGTS	90
a.	Modular Design	90
b.	Menu-driven Format	91
c.	Tool Input and Output	92
d.	TGTS Database Design	92
2.	Tool Operations with TGTS	95
a.	TGTS Database Output Operations	95
b.	TGTS Test Goal Update Operations	97
c.	TGTS PSDL Update Operations	100
d.	TGTS' Iteration Information Facility	101
C.	USE OF THE TEST GOAL TRACKING SYSTEM	101
1.	Problem Environment and Prototype Specification . . .	102

2. Message Processor Requirements	103
3. Test Goal Development Process with TGTS	108
D. PERFORMANCE OF THE TEST GOAL TRACKING SYSTEM	111
V. CONCLUSIONS AND RECOMMENDATIONS	112
A. RESEARCH CONTRIBUTIONS	113
E. FUTURE RESEARCH	114
APPENDIX A TGTS DATABASE SAMPLES	116
APPENDIX B TGTS USER'S MANUAL	138
APPENDIX C TGTS SOURCE CODE	170
REFERENCES	276
BIBLIOGRAPHY	279
INITIAL DISTRIBUTION LIST	282

LIST OF FIGURES

Figure 1. Iterative Rapid Prototype Development	7
Figure 2. Waterfall Life Cycle Model	8
Figure 3. Rapid Prototyping Methodology	9
Figure 4. Conventional Test Flow	16
Figure 5. Spiral Model of the Software Process	53
Figure 6. Main CAPS Tools	57
Figure 7. Shimeall's Iterative Test Planning Process	68
Figure 8. A "Targeted" Spiral	74
Figure 9. Top Level TGTS Program Decomposition	91
Figure 10. TGTS Database Structure	93
Figure 11. C3I Message Processor Module Decomposition	104
Figure 12. Manage Radar Tracks Sub-module	106
Figure 13. PSDL Description of Radar Track Manager	107

ACKNOWLEDGMENTS

I wish to express my gratitude toward, and admiration for, Professor Timothy J. Shimeall for providing the initial idea for this thesis and for his mentoring efforts. He introduced me to the fascinating field of software testing and has been a phenomenal source of computer science knowledge.

Captain Patrick D. Barnes, USAF and Professor Luqi have also been a regular source of professional assistance. I hope their continuing research efforts will be enhanced by the work presented herein. It has certainly been my privilege to have been associated with them.

My wife, Janet, and our three boys, Micah, Matthew and Mark truly have been a blessing and a regular source of joy and encouragement throughout the process of thesis preparation. Mark arrived in the midst of the process and occasionally kept later hours than I did. Amidst the hustle and bustle, their prayers and help around the house were a great asset.

Finally, if anything shows the fallibility of man in his fallen state, it is software testing. Hopefully, this thesis' results will improve software quality and performance. Sola Dei Gloria.

1. INTRODUCTION

Prototyping is a rapidly developing software engineering paradigm that is receiving increasing attention as a way of speeding software development. As we look to rapid prototyping methodologies to evolve production code from prototypes, the need for effective testing methodologies becomes increasingly important. Current testing approaches are not designed to support evolutionary or iterative prototyping methodologies. Testing in a prototyping context has been either ad-hoc or else has failed to accomodate iterative development. Shimeall's work [Shimeall90] initiated questions regarding testing for iterative prototyping. This thesis will describe a software testing approach for evolutionary iterative rapid prototyping systems. The following sections overview issues necessary to fully introduce the implications of testing support for prototyping.

A. SIGNIFICANCE OF RAPID PROTOTYPING

Rapid prototyping is among the most promising emerging technologies in the field of computer science today. It facilitates a systematic and automatable means of developing a desired software system under circumstances where initial requirements are not well known or where requirements change frequently during development.

1. The Need for Prototyping

a. The Software Gap

The past two decades have seen the gap between software system demand and development expand into a backlog known as the "software crisis." Software development has not effectively kept up with demand. Computer hardware technology has outpaced our ability to develop software that fully exploits the hardware's capabilities and meets the user's stated requirements on time. Many reasons for the backlog exist. The techniques for engineering requirements for large, complex systems are still immature. This immaturity exacerbates resource limitations and lengthens development times. Requirement and hardware changes are also a certainty during large system development that, in turn, will require system changes. Increasing software development costs mitigate toward extending the service life of software, therefore software should be developed within a life cycle that eases system maintenance and modification throughout its life. Ensuring that systems are amenable to modification also slows development. System size and complexity today demand automated tools for system development and flexible development paradigms. Both the tools and the paradigms should be adaptable to change throughout the development process.

b. Software System Reliability and Maintainability

Reliability and maintainability are the other key system development problem areas today. The key question is, "How does one know that a system is correct and does all that is required and nothing that it is required not to do?" The larger the system, the more complex one's

attempt at an answer will be. In short, test teams must test the system for proper behavior. Testing methods must give the necessary assurance that the system works. Debugging must be quick, complete, and consistently applied where it propagates change throughout a system. Current manual methods are slow, tedious, and error prone. Maintenance is a very broad area that includes "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt a product to a changed environment" [ANSI Stds83]. It currently accounts for over half the total software cost in today's systems. Maintainability implies a need to track the adjustments to a system over time and the accompanying verification and validation that come both with the initial acceptance and with each system change. Even more fundamental, the system and its life cycle model must promote easy change.

2. Purpose of Prototyping

a. Prototyping as a Concept

Prototyping is not a new concept. Industry has long applied prototyping as a means of determining project feasibility and model refinement. Aircraft prototypes are an example of this. Prototyping is a new concept though, within the realm of software engineering.

Most software prototyping work to date has been with prototyping for requirements analysis purposes. A model (prototype) of the anticipated software is constructed for customer and developer assessment. In some cases (i.e., real-time systems), a model is the only effective requirements derivation technique. Once the model is built, it is demonstrated to the user for his feedback. The "develop, demonstrate,

modify" approach is repeated iteratively until all requirements are formalized or until the system evolves into a production system.

Pressman notes that prototyping has been conducted with one of two objectives in mind:

- the purpose of prototyping is to establish a set of formal requirements that may be translated into production software through the use of software engineering methods or techniques, or
 - the purpose of prototyping is to provide a continuum that can lead to the evolutionary development of production software.
- [Pressman87: p.150]

The two purposes are not necessarily mutually exclusive. Methodologies can be used both to capture requirements and subsequently to build a model or instance of a system to serve as a pattern upon which to iteratively develop later stages of the system. Prototyping is useful for capturing system requirements when they are not well known or when they are of questionable feasibility. A prototype provides a partial representation of a desired system to be used in system analysis and design to capture requirements and/or determine feasibility. One can apply iterative rapid prototyping to the spiral model-iterative design approach to provide a pilot version or executable model of the intended system and evolve (transform) that model into the production system. Prototyping then becomes the software engineering technique used to transform requirements into the production system.

Researchers have proposed several software prototyping methodologies to which the preceding paragraphs have alluded. While authors have coined many names for these methods, they can be taxonomized into five basic methodologies.

1. Rapid Throwaway Prototyping Methodology. This approach augments traditional system development methods by providing a quick implementation for users to test and then provide a response to the designers on how well it meets requirements. It is used before or during the requirements phase of the life cycle to enhance requirements capturing and does not become part of the production code.
2. Incremental Development Methodology. A partial model of the system is developed and then enhanced in small, successive increments. The target here again is capturing requirements without using the resulting prototype in the production code.
3. Evolutionary Prototyping Methodology. Methods one and two are combined to capture requirements and then iteratively develop the system. The resultant prototype is used as part of the production system. The prototype is "tuned" along the way and proponents of this methodology see it as a way to eliminate formal, written requirements specifications.
4. Reusable Software Component Methodology. This methodology builds prototypes using written components retrieved from a library, thus saving development time. The components must be written in a common language that allows for an adequate interface after component linkage. Portions of the prototype may become part of the production system.
5. Automated Software Synthesis Methodology. This method transforms high level design specifications directly into operational code so that the prototype attains full functionality via full specification. The specification serves as the source to generate the operational code and the specification language provides the means for all the design, coding, and integration. This method is not currently feasible.

A thesis by Fountain discusses these methods at length [Fountain90]. Chapter II analyzes each method in detail, particularly noting which methods need formal testing support. Formal testing methods are applicable to those methodologies that can directly result in production code development.

Rapid-prototyping is a key descriptor for the concept of software prototyping since developers need ways to speed the software design process. Prototypes must be capable of being developed rapidly for

prototyping to be effective. Pressman cites three generic classes of methods and tools used for rapid prototyping: fourth generation techniques, reusable software components, and formal specification and prototyping environments [Pressman87: p. 150]. Rapid prototyping exists specifically to help create software in instances where requirements are not well known or are not well understood or may change often and readily adapts to requirements changes. Evolutionary and iterative rapid-prototyping systems capture functional requirements via a series of increasingly functional partial implementations to produce a system that matches the customer's desires after development. The customer evaluates each partial implementation which is then extended to reflect his comments. Figure 1 graphically illustrates this process. The spiral suggests the iterative prototype development, with the angle denoting the developmental phase and the radius indicating a measure of software completeness [Shimeal190: p.1].

b. Prototyping Contrasted with the Waterfall Life Cycle Model

Prototyping, when applied iteratively within a spiral life cycle model, is flexible and responds quickly to the user's feedback. This is in sharp contrast to the Waterfall Life Cycle Model of software development where all requirements-capturing occurs as the first step in a sequential process of system development. Developers must "lock in" requirements at the onset of development or else face the prospect of requirements changes that force them to back up in the development sequence and reimplement the changes. The sequential nature of the Waterfall Model makes it unresponsive to change and a poor validation

method for uncertain software system requirements. Figure 2 graphically

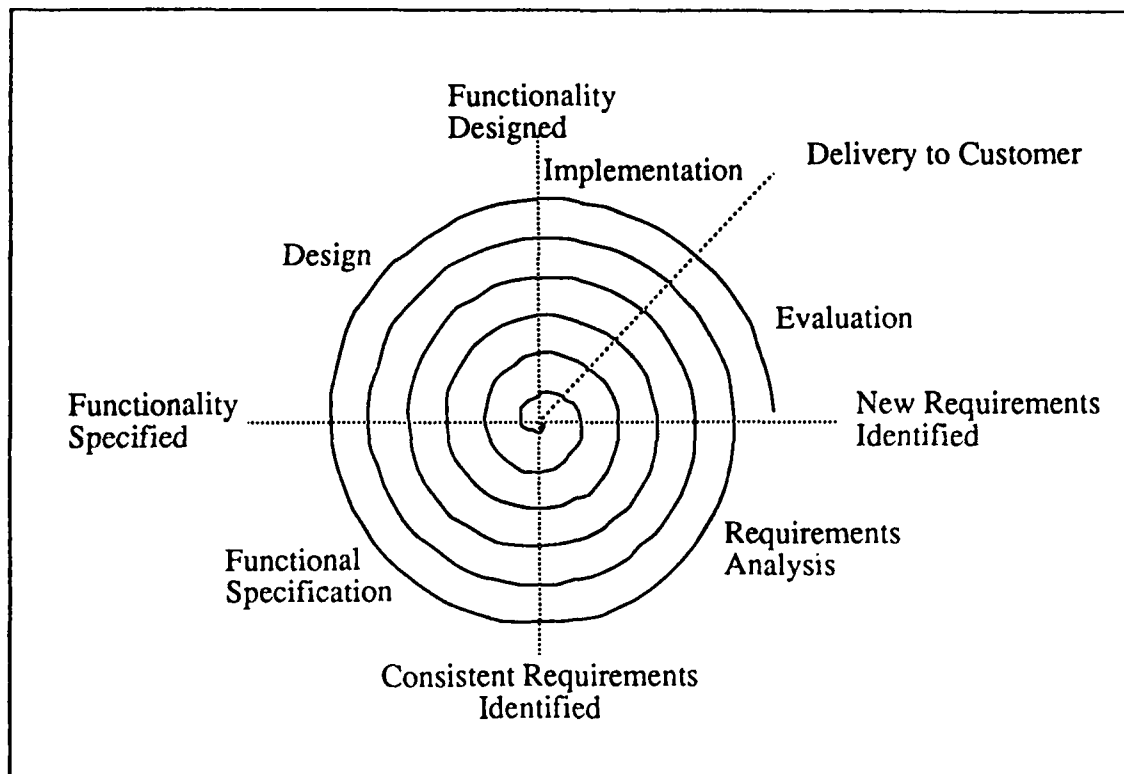


Figure 1. Iterative Rapid Prototype Development [Shimeall90: p.1]

depicts the Waterfall Model as described initially by Royce in 1970 and as referenced by Davis, Bersoff and Comer [Davis,Bersoff&Comer88].

Though some claim that prototyping is too expensive, studies suggest that it is still far cheaper to prototype than to correct a system after it is in production. One can readily see that it is much simpler to correct errors in requirements and specifications early, rather than once the system has gone into production. Gomaa and Scott were among the first to point this out as far back as 1981 [Gomaa&Scott81].

Specifying user requirements has always been difficult. The written specifications typical of the Waterfall Model are generally dull

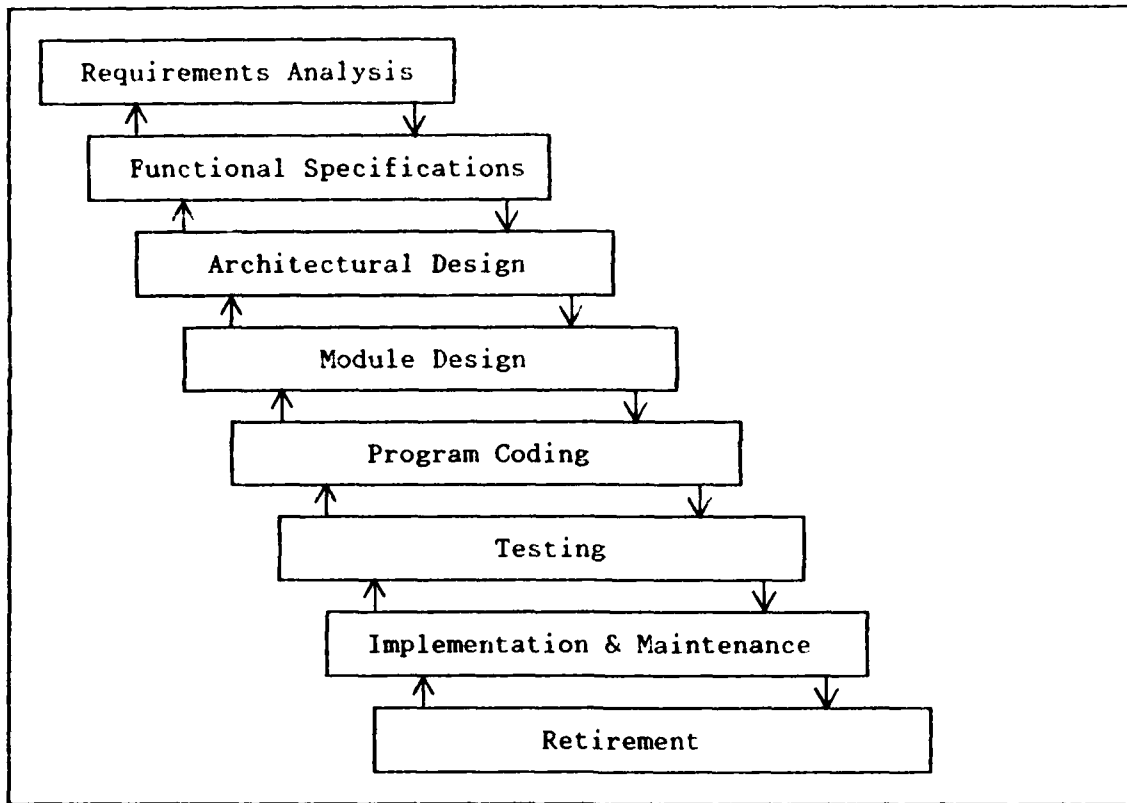


Figure 2. Waterfall Life Cycle Model

reading and difficult for users to understand. All this presupposes the designer correctly captured the user's requirements in the first place. On the other hand, prototyping provides a working model (or prototype) that makes the designer's interpretation of the user's functional requirements clear and helps identify misunderstandings, missing requirements, and errors. The use of a prototype clarifies and verifies that the designer correctly captured the user's functional requirements. A functional requirement is "a requirement that specifies a function that a system or system component must be capable of performing" [ANSI Stds83]. By taking the user's comments on any iteration, designers make appropriate corrections and additions to increase the prototype's functionality.

Figure 3 shows the standard rapid prototyping methodology using a feedback loop.

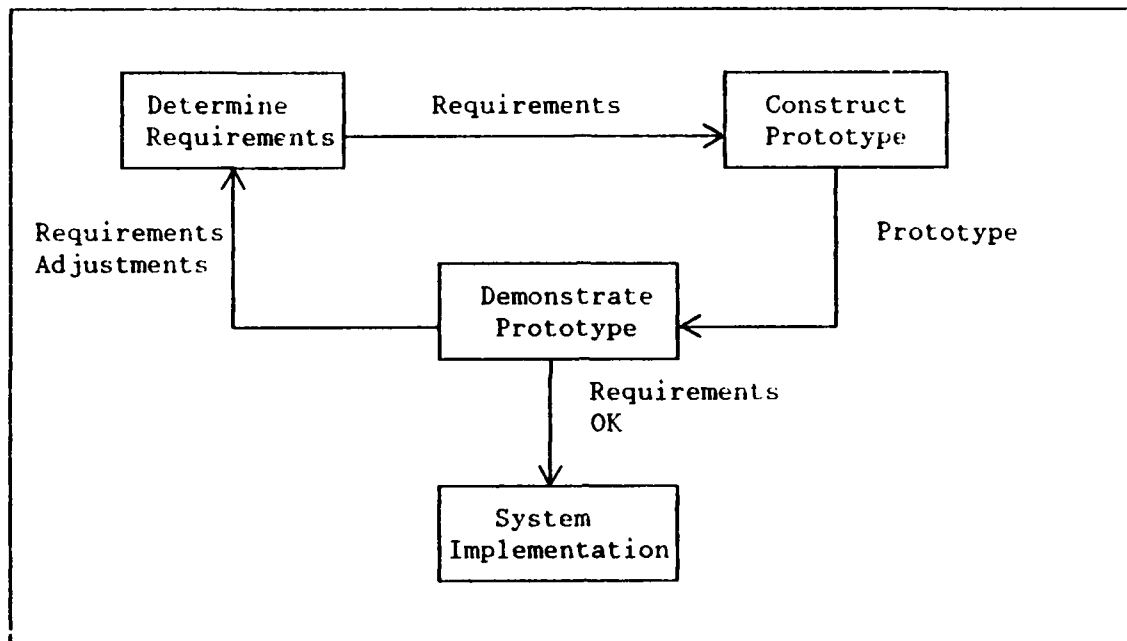


Figure 3. Rapid Prototyping Methodology [Fountain90: p.16]

3. Techniques of Rapid Prototyping and Testing Implications

Evolutionary rapid-prototyping methods are especially interesting since they lead to production system development directly from the prototype. Production code is the object of formal testing. Evolutionary prototyping incorporates the incremental (or iterative) methodology and is efficient because it uses the prototype directly in producing the delivered (production) system. Using such an approach over the system lifecycle closely parallels the spiral model of systems development. Boehm describes the effectiveness of the spiral model and its adaptability to prototyping [Boehm89]. This thesis investigates the implications of the spiral model on prototype testing.

Coupling rapid-prototyping with capabilities such as iterative development and libraries of reusable software components, speeds development since a prototype demonstration will show quickly whether the system meets the customer's requirements. However, prototype demonstration alone is not generally sufficient evidence for determining that a software system performs acceptably. Therefore tools to aid in software testing form an essential part of a production-system-oriented rapid-prototyping system. To generate production code directly from prototyping will require evolutionary, iterative rapid-prototyping environments, employing reusable software components. These characteristics show great promise for the development of production software systems, to include very large applications in real-time computing environments using high-level languages.

4. Strengths and Weaknesses of Prototyping

As mentioned, the current Waterfall Model of lifecycle development is proving to be too slow and costly for today's large software systems. Prototyping promises to reduce development costs and time for several reasons. First, prototyping puts the onus of requirements validation on the user, who must validate each iteration of the prototype during demonstration. Second, requirements tend to be captured earlier because users comment the system after each iteration vice at the end of implementation, therefore erroneous or missing requirements are less likely. The expense of requirement errors is a major development cost. Third, all prototyping methods allow requirements validation to be completed before production and provide working models

that are available for evaluation throughout the development process. Prototyping also promises to produce equivalent software faster than conventional approaches [Boehm84: p.86].

Several weaknesses of the prototyping methodology bear mentioning as well. First, there is a tendency not to record requirements as they are captured. The argument for not recording requirements is that the working prototype and the prototype specification language provide the requirements. Such a view poses problems because it does not address the need for those not expert at reading prototype requirements (customers) to understand the system requirements apart from watching a demonstration. Failure to record requirements also does not address the need for formally testing the prototype. Much of the intent of a prototyping specification is implicit and not readily evident. Prototyping captures the "what" of a system, but often loses the "why." Justification of the system's behavior is therefore often hard to state, especially as the size of the system grows.

Another weakness of prototyping is its immaturity as a methodology. The diversity of prototyping methodologies under development and the lack of completed sophisticated prototyping environments limit its current applicability.

5. Prototyping Today

As mentioned, prototyping is a new field and no general purpose prototype systems (environments) have been fully developed, so many environment implementation questions are not yet answered. Increasing the prototype methodology complexity elevates the need for complex

development environments to support the prototyping capabilities. Most of the research in evolutionary and reusable software components methodologies is likely to require five to fifteen years to mature since these prototyping environments are much more complex than the simple user interface demonstration methodologies in use today. Maintaining design histories and matching requirements and their modifications to prototype iterations is another major research task needed for building effective prototyping environments. Fountain provides a detailed analysis of the existing methodologies and published models [Fountain90].

B. SIGNIFICANCE OF SOFTWARE TESTING

Software systems abound today and the list of additional application areas continues to grow rapidly. As software components control more life, mission, and cost critical systems, we must be the more sure that these systems are complete and correctly designed. Software systems testing verifies that these systems are complete and sufficiently correct as implemented. Testing is an essential and complementary partner of design within the software system lifecycle. As the importance of the consequences of system failure increase, so does the importance of software testing.

1. Purpose of testing

"Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results" [ANSI Stds83]. The primary goal of testing is to ensure that bugs do not enter the system under consideration. Testing

accomplishes this goal by making potential mistakes and misunderstandings visible and by identifying incomplete requirements and design. A secondary testing goal is to find the bugs that have managed to enter the system. Testing accomplishes this second goal by causing them to produce a result that conflicts with a specified or expected result [Beizer83:p.3]. Note that neither of these goals is a demonstration that the system will work for all cases.

Systems under development will contain errors since fallible humans are necessarily involved in the development process. Beizer cites the disconcerting fact that programming done well still has on the order of one bug per 100 statements [Beizer83]. Testing must be performed if we are to hope for better system performance than such a bug rate will provide.

Testing and quality assurance are the single largest costs in software development and typically consume thirty to sixty percent of total development resources (more as the size of the system grows). In the early years of computer science, virtually all testing was learned in the school of hard knocks and conducted in an ad-hoc manner. Indeed, it was not until the early 1970's that software testing received serious formal attention [Hetzel88]. The need for effective testing methods has increased with the growth in application size, complexity and cost. Therefore, efforts have expanded in the software engineering community to formalize and systematize testing methods. Today's system complexity mandates that a design system and its lifecycle model be highly amenable to thorough testing with minimal resource cost and maximal assurance of proper system performance.

2. Conventional testing and its limitations in prototyping

There exist theoretical limits to complete testing. Testing is never a proof of program correctness. Any given program will operate on a finite set of inputs and complete testing would subject a program to every conceivable input. Therefore complete testing is practically impossible due to the combinatoric explosion of possible inputs for a program of any size. Beyond this, Manna and Waldinger succinctly express the theoretical limits to complete testing:

- "We can never be sure that the specifications are correct."
- "No verification system can verify every correct program."
- "We can never be certain that a verification system is correct."
[Manna&Waldinger78: p.199]

Therefore the objective of testing is to demonstrate suitability as opposed to proving correctness.

How one defines suitability is context-sensitive and must be determined within the framework of the application environment. We want a level of assurance that the system is correct commensurate with the criticality of the system. Therefore the need for effective testing methodologies arises, ones suited both to the development and the application environments so that we can declare a system performs suitably. Concurrently, testers need ~~automatable~~ testing methodologies to complete testing in an acceptable time.

Testers need three things to conduct software testing:

- an object to test (e.g., a system containing software)
- a set of test conditions (e.g., data to feed through the program)

- a standard for evaluating the behavior of the object (e.g., a statement of correct program results) [Shimeal190: p.2]

Given a system that needs testing, testers collect test conditions and develop a test oracle against which to measure the system's suitability. Testing unfolds in the following manner. The tester collects and validates the needs and assumptions made during the application development and then expands and develops them into a set of consistent test goals in a manner similar to requirements analysis.

Based upon a test goal analysis, testers state and prioritize possible test goals. Since only a fraction of the possible inputs of a system can be tested, the tests must be prioritized and then grouped to ensure the most efficient testing and to know what to abbreviate, if required. The testers select specific test conditions to satisfy the test goals. A single requirement may require multiple test goals. Each test goal may additionally require multiple test cases to ensure that the proper behavior is present. Testers will use various testing techniques to select test cases and evaluate the system under consideration. They place tests in a logical order to test particular goals and situations, with progressively larger portions of the system being tested as the process continues.

At any given level, testers test the most critical portions of the system first to make sure that those portions perform correctly. Once testers derive the test oracle, and the user approves the tests, the tests will be executed and analyzed. System prototypers and testers correct encountered test irregularities in the prototype and the prototype test plan respectively. Testers generate additional test conditions and tests

for the next iteration of testing, if needed, to test previous corrections. Figure 4, from Shimeall [Shimeall190: p.3] provides a picture of the conventional testing process.

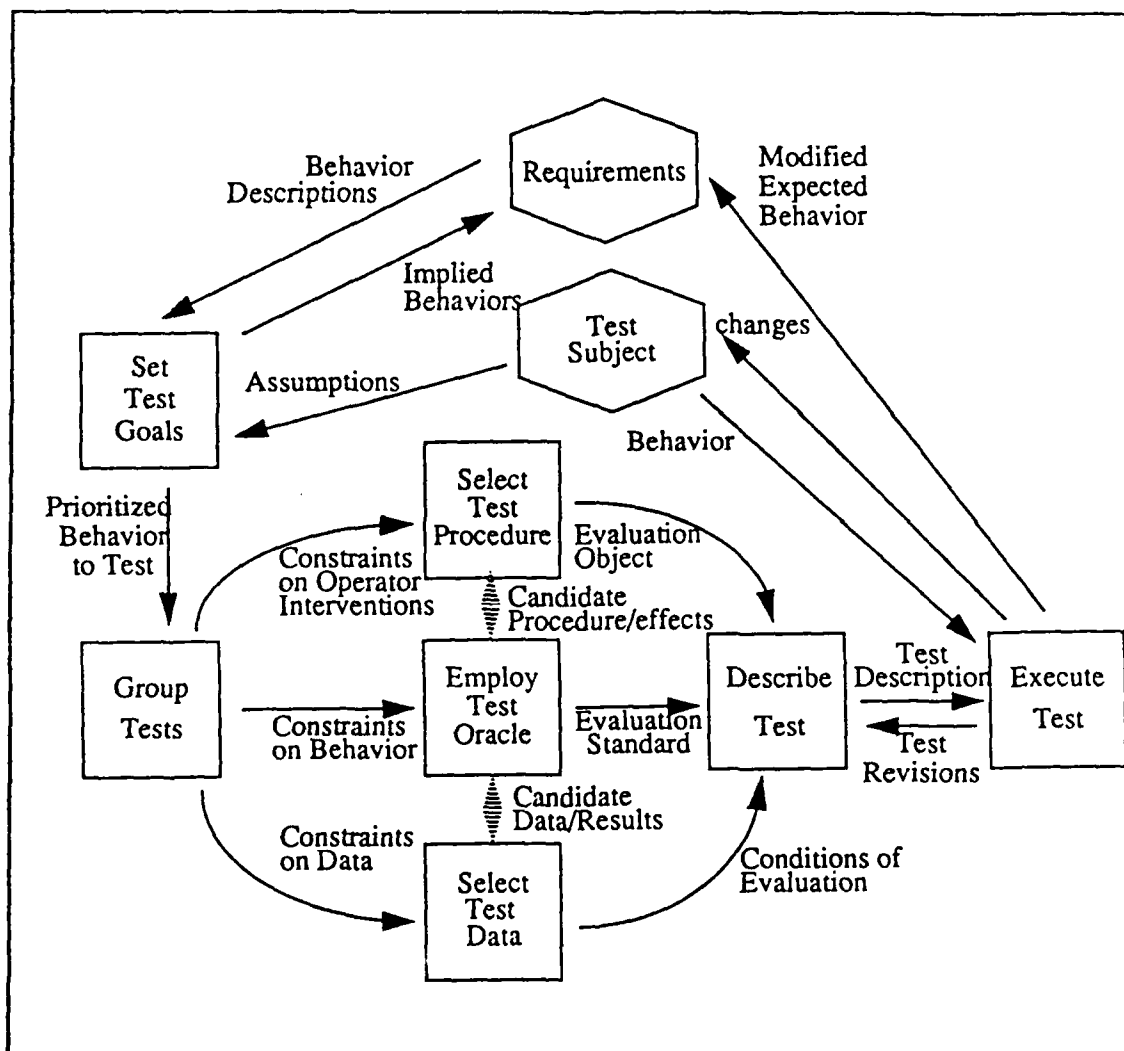


Figure 4. Conventional Test Flow [Shimeall190: p.3]

Prototyping brings some unique testing concerns as well. One iteration of a prototype can vary significantly from another. Requirements may be changed or added or deleted. As iterations progress, the number of requirements explicitly added and the requirements that will

need to be derived due to requirement interactions will increase significantly. What worked in the system's context in one iteration may well not work in the context of some future iteration. When using reusable components, changes in component context of use will usually require further component testing. Changes in the prototype specification also may change the component chosen in reimplementation, thus instantiating different code to match the changed specification. This implies that prototype testing must occur in the context of the prototype's history, the current set of requirements, and any previous testing conducted.

While prototyping languages capture **what** the prototype is to do, they are weak at capturing **why** the prototype is to do this and the accompanying assumptions. Without capturing requirements, building a proper test will be virtually impossible in contrast with conventional testing where such documentation exists as part of the waterfall model of development. All the above cries out for a companion testing methodology that promotes a high assurance of proper system behavior and is tailored for rapid prototyping.

C. PURPOSE OF THESIS

The purpose of this study is to investigate software testing within an evolutionary, iterative-rapid-prototyping environment, and to establish both the need for and feasibility of a companion testing methodology for iterative rapid-prototyping. Testing support is a critical aspect of such an environment since the prototype will evolve into the production code. This topic has not previously been investigated. This thesis also

investigates methods for software testing tool development for integration into rapid-prototyping environments. Now is the time to consider how to enhance the rapid-prototyping methodology to facilitate thorough software testing. Once prototyping allows thorough testing, we will have a powerful means of developing very economical software systems with a low risk of system error. Due to the current "software crisis," we must establish more rapid, flexible and reliable software development methods.

Since rapid prototyping, as currently viewed, does not result in any final statement of the customer's expectations, apart from the series of prototypes, there exists no objective standard as a basis for testing the software product. We need a formal testing methodology to accomplish such testing. Therefore, formulating a corpus of foundational thought for implementing specific testing methodologies on particular qualifying rapid-prototyping systems will allow us to build testing tools for them.

A second purpose of the thesis investigates how to conduct requirements-based testing within rapid-prototyping environments. Since a series of prototypes results in a form of the customer's system requirements, a central aspect of any functional testing must include capturing these requirements in a way that leads to automatable requirements-based testing methods. These methods must remain consistent with rapid prototyping's purposes and concurrently move us toward increased testing reliability and speed. Such methods also should aid in executing a formal test process cycle.

A supporting thesis purpose is to demonstrate software testing tool feasibility in rapid prototyping by constructing a simple requirements-based testing tool for use in a particular rapid-prototyping system. This

tool will be the first member of a planned family of increasingly capable rapid-prototyping testing tools.

The foundation for building a rapid-prototyping requirements-based testing formalism exists in the necessary testing information's residency within the development process and in our ability to usably capture this information for testing purposes. The thesis will show, from key rapid-prototyping system characteristics, such as system design, prototyping language, revision control, and implementation language, how each contributes to effective tool design. It must then establish key testing tool design principles supporting the rapid-prototyping environment. Finally, the thesis will provide recommendations for continuing research in rapid-prototype testing.

D. SCOPE OF THESIS

The entire spectrum of software testing concerns accompanies the software testing in rapid prototyping issue. Shimeall raises the topic of iterative prototype testing and asks key questions on the topic [Shimeall90]. This thesis elaborates on prototyping methodologies best suited to lead to production code requiring testing. Prototyping methodologies, lifecycle models, and current testing practices are investigated as appropriate, to clearly establish an effective testing methodology supporting rapid prototyping. The need for a prototype testing methodology is highlighted in the following section. The requirements-based testing tool developed will be demonstrated, by way of example, to substantiate the testing methodology presented and recommendations and conclusions regarding the research will be presented.

1. Need for a Companion Testing Methodology in Rapid Prototyping

As mentioned earlier, a given software design/ implementation method is only as good as the attendant ability to test the resulting system sufficiently. Such is no less true for the rapid-prototyping process. One has a frustrating situation, at best, if prototypes can be developed quickly, but then require most of the allotted development time for testing to see if they really meets requirements.

Most prototype testing proposals to date, where such proposals exist at all, involve only manual procedures and ad-hoc testing strategies. To achieve this thesis' primary goal, an essential set of principles and characteristics for building effective rapid-prototyping testing tools must be developed. In particular, the thesis will concentrate on tool development that allows requirements-based testing in rapid-prototyping environments.

Ideally, a given system's design process can take place in a development environment that uses a fully integrated design tool set. This is no less the case in rapid prototyping. In fact, one might argue that such tools are all the more essential with rapid prototyping, since a convenient set of integrated testing tools will aid the primary goal of rapid development. Rapid prototyping's complexity generally requires an environment of integrated and highly specialized tools. Because of the modular designs typically used, the system is generally adaptable to further tool integration, subject to certain restrictions to be described within the body of the thesis.

The general testing method investigated herein views software testing as that process concerned with ensuring a software application

does all it is intended to do and nothing it is not intended to do. There must be a certain measure of formalism to automate the testing process. This formal testing process is essential in rapid prototyping, both to ensure prototypes meet all validated requirements and to isolate and demonstrate new iteration enhancement effects for receiving necessary customer feedback.

The means of capturing the information believed to be present or else made to be present in the rapid-prototyping process is a major thesis concern. There are many possible mechanisms that may aid this process, to include:

- mechanisms to allow analysts and designers to record assumptions made during analysis, specification, and design, to assist in test goal analysis;
- history predicates, identifying modifications made between iterations of the rapid prototyping development, to provide a basis for decision making during test goal analysis;
- analysis of developer comments on the intent of portions of the specification and design, to provide a basis for test oracle derivation;
- analysis of user comments on the intent of usage during the evaluation phase of rapid prototype development, to provide a basis for test oracle derivation and test condition selection;
- automatic generation of test documentation from captured information. [Shimeall90: p.9]

Since this is the first rapid-prototype testing study, the thesis investigates key principles and characteristics of rapid-prototyping systems that lend themselves to testing mechanisms, with the main effort being requirements-based testing and test goal analysis.

Currently, there exists an evolutionary, iterative rapid prototyping system that is undergoing development at the Naval Postgraduate School. This system, the Computer Aided Prototyping System (CAPS), is intended primarily for development of real-time applications using a reusable component base, currently Ada. The requirements-based testing tool developed in Chapter IV is for use with this prototyping system.

Over the long haul, substantial research will be needed to cover effectively the entire issue of testing within evolutionary, iterative rapid-prototyping environments. This thesis seeks to open the door on the subject.

2. Overview of Thesis

Chapter II provides an overview of the problem domain, covering selected rapid prototyping methodologies that appear to possess the most promise and adaptability for future applications and development of formal and rigorous testing methodologies. A selective overview of the purpose of testing and currently automatable software testing methodologies follows. Chapter III proposes a software testing tool methodology for evolutionary, iterative rapid prototyping systems, using reusable software components. It includes a description of the system characteristics needed for testing that attends particularly to requirements-based testing tool development. Next, Chapter IV describes a basic requirements-based testing tool developed for use with the Computer Aided Prototyping System (CAPS) currently under development at the Naval Postgraduate School, Monterey, California. The chapter describes tool implementation, usage

and performance. Chapter V summarizes the thesis contents and proposes directions for further research.

II. SURVEY OF PRECEDING WORK

Software testing within an iterative, rapid-prototyping environment is an interesting blend of several software engineering sub-disciplines. Resultantly, the topic can be fully discussed only by establishing how software testing and rapid prototyping interact. This chapter discusses rapid prototyping, software testing, current rapid-prototype testing methods, and several other lesser topics pertinent to the research at hand. The survey covers the state of the art and focuses on prototyping methodologies that directly develop production code and on techniques to aid in testing this code.

A. SELECTIVE OVERVIEW OF RAPID PROTOTYPING

1. The Promise and Value of Prototyping

The frustrations that software engineers encountered using the various traditional software development approaches have led them to look to new paradigms to realize the needed software development gains. Prototyping is one such new paradigm that has received increasing attention over the past decade. Many researchers are convinced it holds great promise for helping to solve the software crisis. The prototyping paradigm directly addresses several key problems contributing to the software crisis. Luqi and Berzins summarize the vision of its value as follows: "Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the real needs of the users,

increasing reliability, and reducing costly requirements changes" [Luqi&Berzins88]. This is very much the case since the prototyping process tends accurately to capture requirements early. Further, since it also places the burden for requirements validation upon the user, there is less chance of unexpected changes later. A consequence is lower maintenance cost over the system's life since the user's experience with the prototype during development lowers the chance of performance surprises after system delivery. Where feasibility is questionable, prototyping shows if the system can be developed within cost constraints, without having to commit excessive effort to the project.

The wealth of material written to date on prototyping attests to its many possible meanings. A basic dictionary definition describes prototyping as "an original type, form, or instance that serves as a model on which later stages are based or judged" [AmHerDict78]. In the software engineering context, prototyping refers to an executable model of a requested software system. It typically represents some portion of the entire system, varying in the completeness of its functionality, consistent with the purposes of the particular prototyping methodology used.

The advantages of prototyping in requirements engineering are readily apparent. The two components of requirements engineering are requirements analysis and requirements validation. A requirement, as used in this context, is:

- (1) A condition or capability needed by a user to solve a problem or achieve an objective.
- (2) A condition or capability that must be met or possessed by a **system** or a **system component** to satisfy a contract, standard, **specification**, or other formally imposed **document**. The

set of all requirements forms the basis for subsequent development of the system or system component. [ANSI Std83]

Requirements analysis in rapid prototyping is easier since the user interacts regularly with the designer, especially when development commences. The biggest payoff is in requirements validation since the user validates most requirements early in development, thus reducing the opportunity for ripple effects that propagate through a system when later changes are introduced. In the Waterfall Model, requirements validation by the users did not occur until completed software product delivery. This usually adds to system maintenance costs.

The iterative process is essential to the prototyping methodology's success because it establishes a dialogue between the user and the system designer that captures and validates system requirements. The final validated requirements serve as the basis for the system design. Designers add enhancements to the initial system during each additional iteration and iteration continues until the system is complete and all pertinent design issues are solved.

The above version of prototyping is but a skeleton of the whole vision of rapid prototyping. As summarized in Chapter 1, at least five prototyping methodologies have been proposed to date. Advances in Computer Aided Software Engineering (CASE) tools provide considerable help in automating the entire rapid prototyping process to make it truly rapid and manageable. CASE tools are based upon formal specification languages, therefore, to support the tools, the specification language, any prototyping language, and the design language must be fully integrated to achieve automated performance in a given prototyping system. Current

research promises to provide rapid prototyping systems that produce production quality code for large software systems in a fraction of the time required for conventional development methods. Additionally, prototyping environments promise to be very complex systems requiring many supporting tools such as text editors, database managers, dictionary maintenance tools, procedural language program generators, screen generators, documentation reporters, non-procedural report writers, teleprocessing monitors and interactive query languages [Fountain90]. Each tool itself is a research area.

Published prototype system research has focused on subsets of complete prototyping systems. The complexity of the issues involved in prototype system implementation are sufficiently great that no such system will likely be completed soon. Many issues are polemical, such as the need for specific prototyping languages and the feasibility of a single prototyping system's usefulness to multiple domains. Fountain addresses these concerns in his thesis [Fountain90: pp.137-139].

2. Analysis of existing prototyping methodologies

This section briefly reviews the five major prototyping methodologies proposed and/or used to date. The goal here is to discern prototyping methods that are most likely to evolve into production systems and that will need testing support. Fountain conducted a thorough, in-depth methodology analysis that is summarized in the following subsections and extended to apply to formal testing methodologies supporting the final production system. Fountain considers six evaluation criteria for the prototype methodologies:

- **Prototype development** - the efficiency of prototype construction and its effect on requirements engineering, especially the ease of user requirements validation and change;
- **Use of reusable software components** - the use of reusable components in prototype construction and whether the production coding is manual or use of reusable components is promoted;
- **Evolutionary prototype production** - whether the developed prototype results in a production system or is used primarily for requirements validation;
- **Meeting user needs** - meeting user needs of on-time delivery and fulfilling requirements (Fountain notes that this information is anticipatory rather than currently available);
- **Time, activities and effort** - the relative amounts of each expended in using the methodology;
- **Implementation outlook** - the likelihood of when a fully implemented version of the methodology will be available.

For an in-depth treatment of just the methodologies refer to Fountain's thesis [Fountain90: pp.39-99]. This thesis extends Fountain's analysis to include value of a testing methodology - whether a complementary formal testing methodology would be beneficial to help test teams evaluate a prototype design team's final product.

a. Rapid Throwaway Prototype Methodology

This approach produces a throwaway prototype used for validating requirements. Once the user concurs that requirements are correct, the prototype is abandoned and a conventional system development occurs. Typically a subset of the requirements is prototyped. The methodology is described by Bersoff, Gregor and Davis [BersoffGregorDavis88].

- **Prototype development.** The system prototyped lacks a complete definition, limiting its usefulness and allowing only partial

validation of requirements. The intent of the methodology possibly suits it for modeling key portions of small systems. It is difficult to justify devoting much effort to something that will not be used in the production system.

- **Use of reusable components.** This is not supported.
- **Evolutionary prototype production.** This is not supported.
- **Meeting user needs.** The user's initial requirements are captured earlier and more accurately, therefore, needs should be better met and met more quickly than with the Waterfall Model due to the early validation of requirements by the user.
- **Time, activities, effort.** Bersoff, Gregor and Davis [BersoffGregorDavis88: pp.5-6] found that effort expended in this methodology was greater than that expended in the conventional life cycle model. Too much effort for too little gain is the result.
- **Implementation outlook.** The system is achievable now without reusable components. Mature implementations can and should be available within the next couple years. It is a first generation prototyping methodology and has value in getting prototyping paradigms started, but is not an end in itself.
- **Value of a testing methodology.** The value of a formal testing methodology for rapid throwaway prototyping is very limited since no production code results directly from the methodology. It does no good to formally test code that will not be directly used as part of the final system.

b. Incremental Development Methodology

The incremental development methodology is a more detailed and efficient superset of the rapid throwaway prototype methodology. It starts by representing the most difficult parts of the system first to shape the project and then fills in the lesser details.

- **Prototype development.** The increase in detail will provide the user with a larger set of requirements that will more fully match his expectations. The prototype is not envisioned to become the production system.
- **Use of reusable components.** This is not a part of the methodology, but would be an enhancement.

- **Evolutionary prototype production.** The prototype code is not directly used after the requirements have been validated, so evolution is not envisioned. Considerable effort is expended in developing the prototype that does not apply directly to the production system.
- **Meeting user needs.** This approach is more adaptable to user needs than the throwaway approach because of the multiple iterations involved in validating the prototype. The iterative nature also makes the methodology particularly adaptable to change.
- **Time, activities, effort.** More activity is required initially to establish an architectural design to allow for system expansion as the system adds functionality. Particular modules of the system can be validated early and go into production coding while the remainder of the system is defined. This can speed coding efforts and allows more development tasks to happen concurrently.
- **Implementation outlook.** The methodology can be partially implemented now. The iteration capability requires a much more complex development environment than the previous methodology and probably will take eight to ten years to implement.
- **Value of a testing methodology.** The value of a specific testing methodology is limited for the same reasons as in the throwaway approach. Note that test sets related to validated requirements will be built earlier than in conventional approaches since coding on these requirements will start sooner as well.

c. Evolutionary Prototyping Methodology

The main view here is that the prototype evolves into the production system. All the work on the prototype development builds the production software. The approach continues to be incremental and builds the simpler portions of the system first since they are understood from the onset of development.

- **Prototype development.** It uses resources well since the user validates both the requirements and the production system concurrently. This method appears better suited to dynamic conditions since enhancements and change will be realized more quickly. The major cost is in a sophisticated environment.

- **Use of reusable components.** This is not specifically noted as a part of the methodology.
- **Evolutionary prototype production.** The main feature of the methodology is its evolutionary nature.
- **Meeting user needs.** Each iteration should more closely match the user needs and remains responsive to the user and further enhancements.
- **Time, action, effort.** The initial iteration of the prototype is available quickly since it is built from initially known, simple requirements. Therefore user feedback can start more quickly than with conventional life cycle approaches. The requirements engineering efforts are less complicated up front but the prototype incrementally tackles more difficult issues and must deal with a larger base of code.
- **Implementation outlook.** This will depend primarily on the development of prototyping tools environments. It will likely take fifteen to eighteen years to see these systems fully functional.
- **Value of a testing methodology.** A testing methodology would be invaluable here since the prototype will become the production code, which must be verified. Requirements-capturing and functional testing will need to be formal beyond the scope of simple demonstration of an iteration to a user via a prototype script. The faster the testing ability is, the better. Testing tools can help substantially, but automation of testing tasks will require formality and structured methodologies.

d. Reusable software components methodology

Fountain rightly mentions that this is an "implementation feature rather than an independent process" [Fountain90: p.92]. Let us now view this feature's value via the evaluation criteria.

- **Prototype development.** This feature should be integrated with other methodologies to harness the power of instantiating reusable components to speed code writing and prototype implementation. This gives us a near term approximation of automatic code generation, since the latter is a long term long shot as we now view it. Note that this does not reduce the designer's efforts at all since he must perform the same work to determine system requirements.

- **Use of reusable components.** Exactly! The degree to which the capability can be harnessed will depend on many factors to include: the size of the components library, the generality of library components compared to the specificity of the particular implementation code needed in a given instance, and the power of the environment's component search tools. Current database management systems technology is relation-based vice specification-based. Solutions are achievable but are not near term.
- **Evolutionary prototype production.** The reusable components feature is applicable. The major issue is **how** to use it in a given methodology.
- **Meeting user needs.** Production system delivery times should be reduced since the code generation should be considerably shortened (less manual coding). The degree of automation involved in the retrieval and reuse process will decide how much faster the methodology will be.
- **Time, activities, effort.** Development schedules will be shorter with less new design and code development.
- **Implementation outlook.** This feature is largely dependent on prototyping tools development and will need a solid prototyping language to integrate the prototype definition and coding processes. Probably five to fifteen years will be required to realize the reuse capability. The reusable components libraries can and should be built now so we can experiment and decide what prototyping system and library characteristics lend themselves to automated reuse methodologies and component integration.
- **Value of a testing methodology.** Testing methodologies and tools that support reusable components will be invaluable to track unit tests conducted on components and suggesting further tests within the context of instantiation. Integration and path testing also will need to occur within the context of reusable components and their interaction.

e. Automated Software Synthesis Methodology

The automated software synthesis methodology is the current "dream level" of software engineers' thinking. All system design, coding and integration would occur in a very high level language (VHLL). Many expect the VHLL concept to provide the same or nearly equivalent software productivity advance as that caused by the introduction of high level

languages in the early 1970's. This VHLL does not exist yet, nor is our technological level or theoretical understanding of involved concepts sufficiently mature to implement it. This situation helps promote rapid prototyping because we do know enough to implement prototyping concepts and then apply what we learn to this next step of automatic code generation.

- **Prototype development.** Development would be extremely fast compared to current conventional practices by harnessing the anticipated power of VHLL.
- **Use of reusable components.** This concept will likely be employed to some degree, but most of our reliance for coding will depend upon the VHLL directly.
- **Evolutionary prototype production.** This will occur with very little effort since the VHLL will transport our concepts directly into production code.
- **Meeting user needs.** The changes here are probably the most dramatic anticipated. Generally, any time the user's needs changed, you would directly generate a new system, rather than patch the old one because of the minimal effort required.
- **Time, activities, effort.** Several currently used development processes, such as design, coding, and integration would completely cease to exist, ergo radically diminished effort expenditures.
- **Implementation outlook.** The idea is at least several rapid prototype generations away and will not be realized until several decades into the next century.
- **Value of a testing methodology.** I emphatically believe that some fashion of system validation and verification will always be necessary. All mankind, and the systems they create exist within a fallen and fallible world. Testers will still need to know what the system developed is to do, what is acceptable performance and what must be excluded. They must then provide sufficient assurance the system performs as stated. The more test researchers formalize and automate this process, the more likely developers and testers will develop reliable systems with small resource expenditures.

3. Conclusions on prototyping methodologies

The methodology evaluations indicate that evolutionary, iterative, rapid prototyping methods using reusable components, hold the most promise in the next ten or so years for providing a more efficient software development paradigm. The system development effort involved in producing software by this methodology synthesis appears to have the best cost/benefit ratio. Reusable components retrieval comes closer to automatic code generation than anything now available or soon anticipated. An incremental, iterative development approach makes good sense when one is unsure of requirements or feasibility and when one must design in a fluid environment, responding quickly to change. The desire for efficient resource use highlights the evolutionary methodology's prime asset: directly produced implementation code. Some might be willing to wait for automatic code generation, but evolutionary, iterative rapid prototyping with reusable components can be available within the short term to help solve the software crisis.

B. SOFTWARE TESTING: PURPOSE AND SELECTED AUTOMATED METHODOLOGIES

Software testing is the process of ensuring that a program does everything that it is required to do and nothing that it is required not to do. Testing is essential to every software development process. Software systems will contain faults (bugs) because people make mistakes and people develop both the systems and the associated tools and hardware that work with the system. User reviews will not suffice for checking program correctness. User reviews, while possibly guided by a demonstration script, are geared to capturing requirements and lack the

rigor of thorough testing practices. The range of test conditions is not likely to be demonstrated and checked. Additionally, demonstration cases important to testing may be of little consequence to a user's demonstration and be viewed by the user as a waste of his time.

A fault is "an accidental condition that causes a functional unit (software) to fail to perform its required function" [ANSI Std83]. Bugs (faults) fall into two basic classes: those of commission, in which we introduce something contrary to specifications and those of omission, in which we leave out a requirement. Testers test to ensure bugs do not enter the production system or, when they do, find and eliminate them. Testers prevent bugs from entering the system by trying to "break" the system code and then correcting the faults that allowed the system to break. Testers find existing bugs primarily by finding discrepancies between the system's results at any stage of computation and the true results for that stage. The ideal end of thorough software testing is error-free code. One's confidence in system reliability and suitability therefore depends largely upon our confidence in software testing.

Chapter I provided an overview of conventional software testing. This section covers the test process in detail, and the key testing methodologies and several associated automated testing tools. It focuses heavily on the requirements-based methodology because prototyping is intricately bound to requirements-capturing and seeks to ensure that requirements are captured and functionally demonstrated. Practically, if we do not capture the required functionality, then it will not matter one whit how well the software structure operates. A solid understanding of

testing purposes and methodologies is essential for forming a foundation for testing in a rapid prototyping environment.

1. The Software Testing Process

The process of software testing begins with collecting conditions that require testing. These conditions are behaviors that the evaluated software must possess and perform correctly. Shimeall notes [Shimeall90: p.2] that the testable conditions should be augmented by a collection of all assumptions made during the development process. Assumptions are often a source of bugs due to misunderstandings between the designer and the user. Finding these assumptions can be difficult and Hernandez's thesis [Hernandez89] deals in part with this issue. The tester then analyzes the requirements and assumptions and expands them into a stated set of consistent test goals. The analysis identifies the parts of the software to be tested and the acceptable behavior for each goal. Next testers prioritize the test goals. They can then match their testing effort to the resources available and know what to eliminate if testing must be abbreviated. Testers also try to provide as much order and economy to the testing process as possible.

Once testers prioritize the test goals, they group them to eliminate redundancy and build test procedures to test the goals. Test conditions such as system states, input data, operator action sequence during test execution and any other special values needed for the test must be specified and may be collected to suit a number of testing methodologies. Test condition selection is not usually limited to just one method. Using more methods increases one's likelihood of removing

bugs. The test team then derives a test oracle for each test condition. The test oracle can take a variety of forms but must provide the information used to judge if the system has performed correctly. Shimeall notes that the test oracle is both "a final set of test conditions and an associated set of result evaluation criteria" [Shimeall190: p.4]. This is because the test oracle derivation process may reveal missing, incomplete or unrealistic test conditions. Testers prepare a test description document next and order the test executions. The customer reviews the test plan and comments/approves it. Once the user approves the test plan, the testers execute it. Irregularities are noted and evaluated to see if they are a result of program fault or of a test plan problem. Bugs then get fixed, test plans corrected and further tests conducted to cover the testing missed due to test plan irregularities. This process can continue iteratively until program performance is acceptable. Regression testing follows and ensures that bug fixes actually work and do not introduce new faults and errors. Additional steps such as test debugging and reviews are interspersed throughout the test development process.

2. Software Testing Methodologies and Companion Testing Tools

Testers use multiple testing methodologies because of existing testing dichotomies. Systems can be tested from either a functional or a structural perspective. Structural testing looks at the implementation details of a program: coding style, structure of the code and its modules, control and flow paths through the code, source languages and various coding details. Functional testing takes a black box perspective, asking what the program is to do and checking to see if it meets its

performance objectives. The latter takes a user's view from the outside and the former an interior view of all the details and interactions within the code and of the code with the host computer(s). Both methods are needed and can be very effective for testing, each having strengths and weaknesses. Beizer, reflecting on this, states,

Both methods are effective and both have limitations. Functional tests can, in principle, detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors, even if completely executed. The art of testing, in part, consists of making judicious choices between structural and functional tests. [Beizer83: p.5]

Testers also may test the system code at various levels, to include:

- unit testing - testing an algorithm, module or atomic functional unit such as a function or procedure;
- integration testing - testing the interfacing of the various units of a program;
- acceptance testing - testing the whole system for acceptance;
- regression testing - testing the whole revised system.

The code writers normally do the unit testing and separate test personnel conduct the other levels. Experience has shown that using separate testing personnel helps prevent them from operating with the same (mis)perceptions as those who wrote the code. Functional and structural testing can be applied at any of these levels. The following sections consider these testing methodologies in more detail.

a. **Requirements-based testing**

Requirements-based testing is a subset of functional testing (described in the following section). This section closely follows Beizer's explanation of requirements-based testing [Beizer83]. Testers usually start with testing requirements because the functionality defined by the requirements is the reason the system is needed in the first place. Conceptually, requirements-based testing is very straightforward. Requirements-based testing takes a system's specified requirements and tests the program to ensure that all required functionality is present.

Testers must first determine the **requirements** from sources such as formal specifications, requirements documents, contracts, user interviews or processes such as prototyping. Not all statements will be requirements, nor will all requirements be stated. Single statements may translate into multiple requirements and multiple statements may combine to imply additional requirements (derived requirements).

System requirements generally require restatement as **test goals** to be independent and explicitly testable. Each test goal should be separately identifiable so that it can be specifically referenced using some number or index. Test goals should be traceable to the requirements that they test. Each test goal should be simply stated and describe only a single aspect of the software so that in testing testers can ask, "Is this behavior present?" Often requirements either singly or taken together will imply test goals. Any such derived test goals should be so identified and the justification for their derivation recorded so that testers can trace back to the test goal's origin.

Once testers determine the test goals, they must be classified as to how testers will verify them. There exist four classes of test goals that follow in increasing order of effort required for validation. **Non-testable** test goals are those testers cannot test, such as vocabulary definitions, user action descriptions and generalities like "user-friendly." Next are **inspection** test goals that testers can tell by observation if they are met. Inspection test goal examples include device usage, language usage, screen layout, and documentation. Third are **analysis** test goals which require observation and inference to see if they are met. Analysis test goals encompass variable usage, expression usage, support software usage, and commenting. Last and most difficult to test are **execution** test goals such as program calculation results, value transformations, value maintenance, efficiency, and timeliness which require observation of properties available only during program execution.

Test goal aggregation is the third step in requirements-based testing. Here testers group tests into sets they intend to test together, called aggregates. They also group test goal duplications so they can be tested together. Some test goals must be tested together because of dependencies. Test goals that are independent also may be tested together because one does not affect the other. Finally, there are test goals that can be subsumed by other test goals. For example, if testing test goals "A", "B", and "C" also effectively test "E", then "E's" testing was subsumed by the tests on "A", "B", and "C". Ideally aggregation will require fewer tests because of subsumption and testing overlap between test goals.

Case selection follows aggregation. By considering the reason for aggregation, testers can usually establish some base cases that fit the aggregate. Typical cases should be considered and can often be obtained from examples in requirements documents. Atypical or extrema cases should be selected to test performance on the boundaries of acceptable input. Pathologic cases are also good candidates since they can heavily tax the system. Bug prone cases should definitely be tried. One should note that the tester is trying to see if the system can be broken. He is trying to outguess the developers. By considering how a requirement might be misconstrued, how a design might be mis-implemented, what might cause an infinite loop, or how bad data could be fed into the system (and many other strategies), the tester can derive data sets that are very likely to find existing bugs.

Data selection is the process of picking the particular data to use. Testers pick a flow of processing to be examined, then choose data that will cause that flow to execute. Logically, testers start with the cases that are our primary focus. Strategies exist for selecting numeric data, bit field data, case control field data, and boundary data. Semantic shifts (different terminology referring to the same thing or the same terminology referring to different things) are fertile ground for bugs, so jargon in requirements, strange definitions, processes with identical descriptions and such are good areas from which to make data choices.

Expected results must be generated. These often must be hand calculated. Testers should anticipate possible test outcomes and establish criteria for correct results, incorrect results and abort

results. Abort results are those that will cause test suspension until programmers correct the fault.

Test cases must be **sequenced**. Sequencing may be a consequence of the test planning process thus far, or established by a separate step at this point. The goal is to make test runs as long as possible to minimize resource use. Cases that require special set-up should be grouped together. Testing should progress from the simple to the complex, with normal and typical cases tested before load and boundary cases. If faults exist with the simpler aspects of the system, the changes required might well nullify any of the other testing, so this sequence just makes good sense.

The last step prior to running the tests and evaluating the results is **aggregate sequencing**. Generally this should follow the precedence of test goals. System inputs and early processes should be tested before later processes and outputs. Common processes should be tested before special processes. The logic here is the same as in the previous step. Additionally, there must be a sense of priorities. The most common processing will likely be the most used so often it is the most important to test. If the normal and common do not work, then the special probably will not either.

The result of requirements-based testing is usually more requirements which the user validates, typically de facto, since they arose during demonstrations or as the result of problems detected by designers and demonstrated for user comment. The designers correct the software and the testing process continues until product acceptance. The next more detailed level of testing is functional testing.

b. Functional Testing

Functional testing verifies that a program or subprogram performs each function correctly. The description of functional testing in this section is taken after Howden's tutorial on functional testing [Howden80]. Functional testing's emphasis is on functionality and the operation of the system. All requirements-based testing is functional testing but not all functional testing is requirements-based since any given requirement may require multiple internal program functions to support the requirement. Design-based functional testing verifies that each module performs each function as intended. Code-based functional testing verifies that each code structure performs each function as intended. This is to ask if there is agreement between the code and the design and if the code does what the design says that it should. Code-based functional testing begins to look much like structural testing because it begins to look in detail at the code and its structure. Structural testing is discussed in the following section.

Design-based functional testing is generally similar to requirements-based testing but does differ in several ways. It works from the module descriptions and not from the requirements of the system. Aggregation is not normally needed because what testers want to test is clear and relates directly to the one module. Testers often need to insert extra code to set and view internal variables and determine the internal state of the program. The inserted code must be carefully placed to limit any interactions and problems in code execution. Results may need to be evaluated relatively, rather than absolutely, noting consistency vice correctness of the result.

Testers select data for functional testing using equivalence classes. Testing applies representative data for an equivalence class because every conceivable input cannot be tried. Testers also use illegal value strategies to see how the functions respond to bad data. The intent here is to see if the system can filter out invalid data yet retain valid but unexpected data. Border cases should be accepted. Elements should not fail when subjected to bad inputs, nor should they corrupt other program elements. Data processing should always be correct.

Many functional testing techniques exist. Cause and effect graphs can be constructed to link specified causes (inputs/equivalence classes) to each specification effect (outputs/functions) with boolean operators, annotating the links with conditions. Traversing the resulting graph yields a decision table that can be converted to test cases, to be run.

c. Structural Testing

Structural testing methodologies test to ensure that each significant part of a program's design structure has been exercised and evaluated. Structural testing uses the control and data flow structure of a program as a basis for test construction. It is a standard follow-on to functional testing to ensure that portions of a program not exercised in functional testing are tested. Additionally, it provides a quantitative measure of test coverage. Structural testing holds considerable promise for adaptation into prototype testing methods. Prototype flow graphs are likely structures for automating structural testing tools.

Structural testing methodologies tend to receive more academic attention than functional and requirements-based methodologies because structural testing is a more tractable problem and testers know more about it. Structural testing is not necessarily a more effective strategy than the others. Howden's explanation of structural testing forms the basis for the description of structural testing that follows [Howden81].

In structural testing, program designs are most often modeled as graph structures. Nodes are typically statement levels and the links are the conditions and controls that direct movement between the statement levels. Path sensitizing is the process of finding the significant portions of the code. Testers have two criteria available for conducting structural testing: use of control flows and use of data flows.

Control flow approaches to structural testing may use any of a number of criteria, to include: all statements, all branches, all loops (zero, one, and many iterations), weighted paths, and all paths. Each of these has some distinct strengths and weaknesses.

"All statements" coverage exercises the program to execute each program statement at least once during the test. It serves as the basis for all other methods and, compared to other criteria, is not hard to achieve, except for unreachable code. Its basic reasoning is, "Has each node been visited at least once?" It may detect erroneous expressions but usually does not detect loop or logic problems and cannot reach unreachable code (as one might expect). Since it only executes each statement once, it does not exercise multiply accessed code by more than one access path.

"All branches" (arcs or edges) coverage exercises the program to take every branch at least once during the test. This also implies all statements and, compared to other criteria, is not difficult to achieve. Some erroneous statements may be detected by this method and it can detect contradictory branches and non-branching branches, but it does not detect missing branches.

"All paths" coverage exercises all paths possible from input to output at least once during the test. This method will detect all faults but it is often impractical to do so because the number of possible paths can be far too large for many programs. The use of looping programs makes the number of paths infinite and "all paths" coverage becomes impossible, therefore a continuing area of research is to discover coverage that finds "all paths" faults without requiring the same resources.

"Selected paths" coverage seeks to cover typical and boundary paths as a way of restricting the number of paths that require testing. It effectively reduces to functional testing and serves to show that the boundary between structural and functional testing can be quite fuzzy at times.

"Weighted paths" coverage divides the possible paths through the program into pools and exercises the program by executing a sample of paths from each pool during the test. Weights are assigned to each path reflecting its relative importance and paths are tested in each pool until a particular score is accumulated. The method implies "all branches" and provides an objective measure of coverage, scheduleable testing and trackable testing. It will detect most faults if the relevant paths are

tested, but tends not to find overgeneralized paths, missing branches and random typographical errors.

Structural testing also may be conducted using a data flow criteria approach [Frankl&Weyuker86] that starts where the code defines a variable and checks to the point of variable use along what are termed define-use paths. It classifies variable occurrences as definitions, undefinitions, and uses. Uses are either computation uses, that directly affect computations, or predicate uses, that directly affect the program control flow through a subprogram. By tracing variable uses, the tester can find improper variable uses and make any necessary corrections.

Most structural testing techniques instrument the tested programs to set up the test structures. At times instrumentation can affect program performance, so one must be careful with such techniques. The instrumentation is essential though in structural techniques to provide the required status of variables, their use and paths tested. Code instrumentation is simpler using automated tools such as ASSET which I describe in a following section.

There is a problem with many of the testing criteria. For any criteria short of path criteria, there are trivial data sets that satisfy the criteria but do not detect faults. The tester's challenge is to select criteria and data carefully to stress the code in a way that detects faults.

Structural testing is a multi-step sequential process that includes selecting coverage, selecting data, generating expected results, instrumenting the code, running the test, and evaluating the results. The coverage level should be high enough to detect faults not found by

functional testing. Structural testing occurs at a finer level than functional testing and testers must realize the two approaches' interrelatedness. If testers run functional tests first, then they can see what cases remain for structural testing and then design tests to cover what remains. Testers concentrate on variables that affect predicate outcomes and then select values that force as much remaining coverage as possible. In running tests they look for excluded values, forbidden combinations, overgeneralization and overspecialization in predicates, and missing functionality. Generally, coverage may be evaluated automatically but passing, failing or aborting the test must be evaluated manually. Failure to achieve planned coverage often indicates bugs are in the code.

Research is continuing in structural testing to address some particular concerns. Problems exist with loop progress and termination. When has the code looped long enough? How do testers know what the loop is supposed to do? Has the loop accomplished its mission? Data flow criteria, such as variable "defines", or "define-uses" are not well understood in the presence of certain loop conditions [Frankl&Weyuker86]. Test researchers have not yet resolved the issue of how to treat loops that build structures like search trees. Treatment of infeasible paths is also problematic. All path sensitizing criteria must deal with infeasible paths. Paths may be either statically or dynamically infeasible. Ideas exist on how to formulate criteria to exclude infeasible paths but there has been little success in automating the process. Suffice it to say that there is plenty of work yet to be done.

3. A Brief Review of Selected Automated Testing Tools

Software engineers have developed many automated testing tools both for research and industry. Most testing techniques in the past have been almost entirely manual and have relied on the savvy and experience of the individual programmer. The emergence of automated testing tools and the formalisms upon which they are built is changing this and increasing the speed and accuracy with which systems can be tested today. The previous sections should give the reader at least a small appreciation for the complexity of the testing issues that must be addressed for sound, methodical testing. This section briefly outlines two testing tools: ASSET - a structural testing tool and UNISEX - a symbolic executor. No examples of a good requirements-based testing tool are available, but these two tools give a good sense of the state of the art for testing tools.

(1) ASSET - A System to Select and Evaluate Tests. Asset is a structural testing tool designed for data flow testing. It is interactive and takes Pascal programs as input. It tests individual subprograms in Pascal and works interactively with a user at a terminal. The user inputs test data to exercise a particular data use criteria and continues to do so until the criteria satisfaction or until he identifies enough errors to decide to stop testing. The data flow analysis performed is intra-procedural. Programs must be syntactically correct and must not contain seven Pascal constructs mentioned by Frankl [Frankl87] as well as several identifier names and file names that the tool uses. It also places several arbitrary restrictions on program size. Asset instruments

the subject procedure and produces a graph of the structure for the type of data flow testing conducted. It outputs appropriate comments for the executed data and keeps a tally of the flows executed and the flows yet to be executed. The tester has a choice of path selection criteria involving predicate uses and computation uses of data. Plans exist to enhance ASSET to add heuristics that will help determine what data associations are executable since analysis problems exist in the presence of unexecutable paths [Frankl&Weyuker86].

(2) UNISEX: a UNIX-based Symbolic EXecutor for Pascal. UNISEX provides an environment for testing and formally verifying Pascal programs for a large subset of Pascal and runs on UNIX. It uses symbolic execution to test the program. Kemmerer's and Eckmann's work provides a complete description of UNISEX [Kemmerer&Eckmann85].

Symbolic execution uses algebraic symbols in the place of an exhaustive set of data inputs since this is infeasible for many programs, as discussed earlier. Equivalence classes of inputs can be established by restricting the values each symbol may represent and the program can then manipulate the symbols and the symbolic results examined to see if they meet with the desired results. Symbolic execution is a form of static testing not discussed in the previous sections. Basically, static testing does not require the program to run or else requires it to run in a non-standard way. Howden's "A Survey of Static Analysis Methods" [Howden81] is a classic that covers static testing in detail. The value of symbolic execution is that all the possible equivalence classes for a program's variables can be represented, thus one can verify that a program

performs correctly for all inputs. Pascal required some language extensions to allow for this. Symbolic execution also discovers how input values for a program must be constrained to cause a particular path to be executed.

UNISEX is a flexible tool. It has two modes of operation: test and verify. It also has a debug feature. UNISEX can provide a trace, can save states, provide various displays of the program or path conditions or saved states or variable values. Breakpoints can be inserted and execution can be either manual or automatic in the verify mode.

The primary stated goal of UNISEX is to have a testing and verification tool useful for academic users. Future work will include expanding the subset of Pascal supported by the tool and simplification theorem proving.

The review of ASSET and UNISEX provides a good reflection of the current sophistication of testing tools and technological capabilities supporting testing. They also show that there exists no all-purpose testing tool that effectively covers all testing paradigms. There exists a need to tailor testing tool development to fit particular testing methodologies and keep their design within current theoretical and technological limits. Proper tool development must consider both the needs of a testing methodology and the aforementioned limits.

C. SYNOPSIS OF SOFTWARE TESTING METHODS WITHIN RAPID PROTOTYPING

At the time of this writing, I am aware of only one published work on the idea of testing methodologies for evolutionary, iterative rapid

prototyping [Shimeall190]. In fact, there were very few comments about prototype testing at all in the literature on either prototyping or testing. This is not surprising when one considers that much of the prototyping to date has been of the throwaway variety and rarely has prototyping been used to evolve production code from the prototype.

In current practice, prototyping activity occurs early in the development process to define system requirements. Once the requirements are established, prototyping stops. Coding follows and testing occurs later, independent of prototyping. Boehm's spiral model of the software process proceeds similarly and will be elaborated in the next section. Therefore, conventional testing approaches have been used at the conclusion of prototyping in a manner and place analogous to testing's position in the Waterfall Life Cycle model. Current literature indicates prototyping is moving increasingly into the commercial world and researchers are seeking how best to apply the paradigm and expand its capabilities. Such being the case, now is the time to research testing methodologies that will provide the greatest benefit to prototyping and receive the maximum benefit from prototyping. To the degree that software development methods ease the testing process and that testing methods build upon development methods, testing and the resultant developed system, will benefit. Additionally, researchers must determine what development paradigms will best suit the prototyping process.

Shimeall has addressed the research topic of testing within iterative rapid prototyping [Shimeall190]. As his work provided the impetus for this thesis, the starting point for considering testing methodologies to support prototyping is his insights. Chapter III takes up this issue.

The spiral model was developed by Barry Boehm, working at TRW over the last few years and incorporates the ideas of prototyping and iteration. It demonstrates the value and flexibility of prototyping in the software development process and considers the place of testing in the process. Figure 5 illustrates the spiral model.



Boehm points out:

The primary functions of a software process model are to determine the order of the stages involved in software development and evolution and to establish the transition criteria for progressing from one stage to the next. These include completion criteria for the current stage plus choice criteria and entrance criteria for the next stage. Thus, a process model addresses the following software project questions: (1) What shall we do next? and (2) How long shall we continue to do it? [Boehm89: p.28]

Note that a process model differs from a software methodology that tells us how to move through the phases of development and deals more with mechanics.

The process concept follows from the previous figure. At the end of each iteration of development, if the prototype is sound for future product development and if requirements mandate further enhancement, another iteration commences. If the present prototype is sufficient, then the waterfall approach continues on through to product delivery. The flexibility of the process should allow any of a number of approaches for product completion. Boehm points this out by noting that the model's primary advantage is that the range of options with the spiral model allows it to become equivalent to other models such as the waterfall model or evolutionary model [Boehm89: p.34]. He cites additional advantages of the model as:

- It focuses early attention on options involving the reuse of existing software.
- It accommodates preparation for life cycle evolution, growth, and changes of the software product.
- It provides a mechanism for incorporating software quality objectives into software product development (emphasis on identifying all types of objectives and constraints during each round of the spiral).

- It focuses on eliminating errors and unattractive alternatives early.
- For each source of project activity and resource expenditure, it answers the key question, "How much is enough?"
- It does not involve separate approaches for software development and software enhancement (or maintenance).
- It provides a viable framework for integrated hardware-software system development. [Boehm89: p.34]

Difficulties mentioned include matching the model to contract software, reliance on risk assessment expertise and the need for further definition and elaboration of the spiral steps. Experience to date with the model indicates it is applicable to large and complex systems [Boehm89]. The model's iterative nature holds considerable promise in evolutionary iterative rapid prototyping and will be investigated further in Chapter III.

E. THE CAPS RAPID-PROTOTYPING SYSTEM

The Computer Aided Prototyping System, CAPS for short, is currently under development at the Naval Postgraduate School, Monterey, California, under the direction of Luqi. It uses the Prototype System Description Language, (PSDL), that supports rapid prototyping based on abstractions and reusable software components for building large, real-time Ada applications. CAPS closely matches our view of the prototyping methodology most likely both to generate production code and to need a supporting testing methodology. It is assumed that CAPS, in its final form, will likely be capable of producing production code. CAPS' methodology and availability make it the choice prototyping system for

this thesis' testing tool research. The following subsections cover both CAPS and PSDL more closely.

1. CAPS

Luqi designed CAPS to support complex programming, particularly control systems with hard real-time constraints. This allows the prototyping system to take full advantage of Ada's tasking capability. It seeks to integrate a full set of prototyping tools to automate as much of the prototyping process as possible by using PSDL to integrate the tools. The main system components (tools) are shown in Figure 6. Note that there are three main subsystems: the user interface, the software database system, and the execution support system.

The user interface allows entry of requirements and design information and presents prototype results to the user. It is interactive and aids in guiding choices on prototype demonstration as well. It also facilitates the propagation of design changes.

The software database system contains two databases. The design database contains the PSDL prototype descriptions of the various projects in CAPS. All reusable components have PSDL descriptions and source code stored in the software base, from which they are retrieved. The software design-management system manages and retrieves the versions, refinements, and alternatives of the prototypes in the design database and the reusable components in the software base.

Finally there is the execution support system. This subsystem's translator "generates an executable framework that binds together the

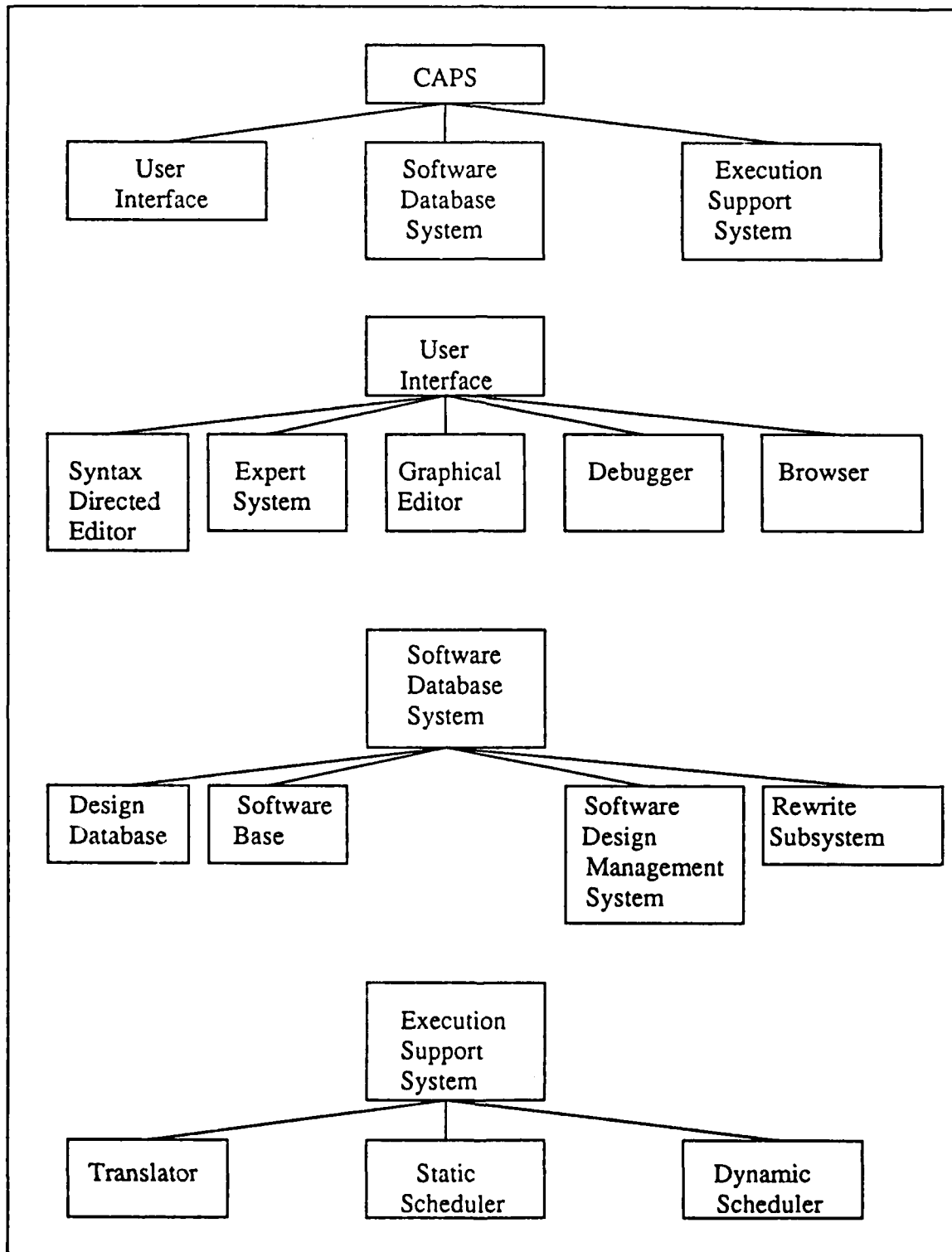


Figure 6. Main CAPS Tools [Fountain90: p.53]

reusable components extracted from the software base" [Luqi89]. It performs functions such as implementing data streams, control constraints and timers. The static scheduler allocates time slots for operators with real time constraints before execution of the prototype begins. Operators are guaranteed to make their deadlines if the allocator succeeds. The dynamic scheduler, on the other hand, invokes operators with no real time constraints into available time slots during execution. Further details on CAPS exists in many publications to include Luqi's [Luqi89] and White's [White89].

2. PSDL

PSDL deserves a bit more attention for it is the glue that holds CAPS together and its power will determine the strength of the final prototyping system. The following information is summarized from Luqi's, Berzin's and Yeh's work [Luqi,Berzins&Yeh88].

PSDL is based on a computational model containing operators that communicate via data streams. Data streams carry values of a fixed abstract data type and also of the built-in type exception. Operators are either data driven or periodic.

Formally, the computational model used is an augmented graph

$$G = (V, E, T(v), C(v))$$

where:

- V is the set of vertices, each an operator;
- E is the set of edges, each a data stream;
- T(v) is the maximum execution time for each vertex (operator);
- C(v) is the set of control constraints for each vertex.

Operators may be either functions or state machines. They may be atomic or composite, where atomic means no more decomposition in PSDL. Composite operators represent data and control flow networks of lower level operators.

Data streams are the communication link connecting exactly two operators, the producer and the consumer. They exhibit the pipeline property and are either data flow streams or sampled streams. Data flow streams are like a FIFO queue where the data is all essential and ordered. Sampled streams are like a holding cell in which whatever is in there gets read when the time is right. PSDL determines the stream type by the activation conditions of the consumer.

Exceptions are built-in abstract data types for creating and detecting exceptions. They can be transmitted along data streams and are encoded as data values to decouple exception transmission from exception handling.

Abstractions are particularly important in prototyping for controlling complexity because they make a system appear simple in order to be prototyped rapidly. PSDL supports data, operator, and control abstractions. PSDL data types are immutable and include ada built-in types, user defined abstract data types (ADT), and those built with PSDL-type constructors. ADT's have both a specification and an implementation. Operator abstractions may be functional or state machine and both have a specification and implementation. The specification attributes describe the form of the interface, the timing characteristics, and both informal and formal observable operator behavior descriptions. They further specify the operator and the basis for retrievals of reusable components.

Atomic operators have a keyword to specify the implementation language. Control abstractions are represented as enhanced data flow diagrams, augmented by a set of control constraints.

Control constraints are either periodic (a period has been specified) or sporadic (no period specified). A trigger condition and output guards are also specified.

Two types of data triggers exist: triggered by all and triggered by some. Every operator must have a period or data trigger or both.

Timers are an abstract state machine that operate like a stopwatch and record the time between events or the time the system is in a state. This allows expression of timing aspects of real-time systems and they are specially treated since they provide a form of non-local control. They remain visible in the module in which they are declared and to the subcomponents of the module.

PSDL supports two conditionals: conditional execution of an operator and conditional transfer of an output. Conditional operators execute using operator trigger conditions. Generally, the trigger condition acts as a guard for the operator and can depend only on either input conditions for the operator or the values of timers.

Exceptions can be produced by both PSDL and the underlying language. Language exceptions not caught by the language must be converted and handled by PSDL. Exceptions can be handled in the underlying language or PSDL exceptions can be converted to the underlying language for handling there.

Stream types are derived by the data trigger of an operator. If a stream is listed as an "ALL" data trigger then it is a data flow stream,

otherwise it is a sampled stream. Operators must be executable whenever their trigger conditions are satisfied.

Timing constraints are essential to real-time systems and PSDL constrains time three ways: maximum execution time, maximum response time, and minimum calling period. Maximum execution time is an upper bound on the time between when a module begins execution and the instant it completes execution and may be applied to all operators. Maximum response time may be applied to periodic and sporadic operators and is an upper bound on the time between the arrival of a new data value and the time the last output value outputs in response to the input (sporadic) or the time between the beginning of the period and the time of the final output. The maximum response time includes all delays while the maximum execution time does not. Lastly, the minimum calling period is for periodic operators only and is a lower bound on the consecutive arrivals of data.

How does it all fit together? PSDL operators are defined in a hierarchical structure, having a number of constraints. Inputs and outputs must fit together consistently, like in building a data flow diagram in systems analysis. The inputs and outputs for connected operators must match, as well as timing constraints, to include "inherited" timing. The execution support system for the language must ensure that all the above constraints are met or error messages are sent. Luqi and Berzins provide some examples of PSDL implementations [Luqi&Berzins88] and the PSDL grammar can be found in work by Kraemer, Luqi and Berzins [Kraemer&Luqi&Berzins90].

III. A PROPOSED TESTING METHODOLOGY AND TOOL DEVELOPMENT STRATEGY FOR RAPID PROTOTYPING

A. RAPID PROTOTYPING SYSTEM CHARACTERISTICS

This chapter assumes the rapid prototyping system will have the following characteristics:

- Iterative;
- Evolutionary;
- Prototyping language with a defined grammar;
- Reusable components capability (library and retrieval);
- Implementation code from reusable components written in a high-level language such as Ada;
- Sophisticated support environment similar to that proposed by Luqi [Luqi89].

These characteristics will keep the prototyping paradigm sufficiently general that we can propose change as required to support testing concerns. Since program verification and validation is the most costly activity in current development, any changes to prototyping to simplify testing will accelerate the development process and increase prototyping's appeal.

B. PROTOTYPE-BASED TESTING

The testing methodology should take maximum advantage of the iterative nature of development. It also should focus on the requirements-capturing purpose of prototyping. Thus, a prototype-based testing technique starts

by capturing the testing information resident in the prototyping process in a form suitable for thoroughly testing the prototyped system. Testers must know both the assumptions and the requirements the designers are trying to meet so that a test oracle and test series can be built to verify the system. Remember, the test personnel are not usually the design personnel, nor should they be, therefore prototype-based testing must provide tools and methods to analyze system requirements and capture requirements changes.

1. Test Information From Iteration

The iterative nature of prototyping implies that the prototyping system must track revision histories and maintain version control of alternate prototype versions. The user's response to a demonstration may require that the prototype fall back to a previous iteration for change or the developer might wish to demonstrate several iteration alternatives for user comment (one of which or portions of several being selected). Requirements may be added (expressed), changed (refined), or deleted. Test goals must easily change to fit modified requirements. Ideally, the prototyping environment will capture this modification explicitly, along with the accompanying purpose of the modification. Any testing tools developed must take these modifications into account to test the proper version and to appropriately consider the requirements and purpose germane to that version for test development. The tool should also exploit change as a likely source of errors. A tool that helps testers compare changes from one iteration to the next, along with system dependencies potentially affected by changes, will help test development considerably.

2. Test Information About Components

The use of reusable components raises reliability concerns about the reusable component library. Have the components been unit tested and, if so, to what degree? Have they been used in previous implementations before, and, if so, which ones? What testing has been conducted on the previous implementations as well as the individual component? The testing methodology must consider how information on past component testing can be recorded and referenced to determine what unit testing might still be needed and what integration testing strategies might best check the components in their instantiated context.

Reusable components use also raises an additional set of test adequacy concerns. Weyuker [Weyuker88] has developed an axiomatic theory of test data adequacy that she demonstrated to be useful in exposing weaknesses in several program-based adequacy criteria. The article also demonstrates that her set of axioms remains insufficient to guarantee test data adequacy. Two axioms, antidecomposition and anticomposition are relevant to testing of reusable components.

Antidecomposition states that testing a program component in the context of an enclosing program may be sufficient with respect to the enclosing program, but is not necessarily so for other component uses. At the simplest level, this implies that just because a component has worked before without a problem, does not mean that it is bug-free in a new context. For example, a context change may introduce interactions that did not previously exist. Previously reachable code may become unreachable or vice versa and error-producing values previously blocked by previous contexts may now be introduced into code.

The anticomposition axiom states that adequately testing each individual component is not necessarily sufficient to test adequately the entire program. This assertion contrasts markedly with some who would suggest that if the reusable components are bug-free then any prototype built from those components will be bug-free. Interactions arising from composition may introduce bugs that were impossible in isolation. If structural testing is considered, then component composition may increase the number of potential paths through the composed code. This might require additional test data to exercise important paths beyond that required to exercise the independent components. Explicit capture of design decisions aids in the determination of important paths.

3. Test Information About Performance

One necessary testing component is a set of test conditions. Requirements-based and functional testing base test conditions upon some stated form of behavior or required performance standard such as formal specifications or a requirements document. The prototyping methodology does not provide a separate performance standard. The testing methodology must establish an objective standard of the intended behavior of the prototype under consideration. Every test involves comparison with an oracle, so every program must have an objective performance standard. The prototyping system must then, in some fashion, provide the tester and his tools with access to a system functionality description and system requirements to allow rapid, complete and consistent derivation of the test oracle from the prototype. This access to functionality descriptions and requirements has the added benefit of helping develop prototyping

scripts for demonstrations so that particular iteration changes and enhancements will be highlighted for the user's comments.

4. Recording Test Information

The prototyping environment should not only capture requirements, assumptions, and design decisions but ideally would map these into the prototype in a way useful to both prototype development and testing. This mapping automatically provides a trace, documenting the prototype's development. As the size of the system grows, knowing why a particular design decision was made and being able to see where (and how) the prototype implements it will be difficult without automated support. The prototyping/testing paradigm must capture mappings from design or prototyping decisions to the implementation. These mappings need to be rapidly revisable to quickly account for prototype changes between iterations.

C. OVERVIEW OF A TESTING SUPPORT SYSTEM FOR EVOLUTIONARY, ITERATIVE RAPID PROTOTYPING

This thesis refines and extends a proposal [Shimeall90] to conduct testing within a rapid-prototyping context. In this section, I summarize Shimeall's proposal for testing within iterative rapid prototyping, describe testing-related features that a prototyping system needs in the light of the prototyping methodology, and then integrate this into describing a prototype testing methodology.

1. Shimeall's Testing Within Iterative Rapid Prototyping

Shimeall's proposal [Shimeall90] provides a framework for iterative prototype testing. Shimeall notes that the most obvious

approach to testing during rapid prototype development would be to treat each development iteration as one software life cycle. He cites as an advantage that this keeps intact the methodology of testing familiar to most testers. He then states the lack of a conventional specification effectively removes the information basis for test planning. Under current descriptions of the prototyping process, a specification would need to be generated, at least in part, before conventional techniques could be applied. The process' complexity is also compounded by the need to conduct a full cycle of testing for each iteration, even though the early iterations will almost never contain detailed or unchanging requirements. This would be inefficient and impractical testing.

Shimeall's alternative [Shimeall190: pp.5-8] is to iterate test planning in parallel with the prototyping iterations. He contends that this will simplify testing and reduce overall testing costs when compared to the above approach. The initial test plan would only consider the basic system description contained in the initial prototype iteration. As prototype iterations proceed, the test plan would expand to incorporate the latest iteration modifications. One disadvantage he mentions is that this approach causes the test plan to follow closely the prototype development. The decisions in the prototype development might unduly influence the test plan, causing important test conditions to fail to be explored. The possible disadvantage suggests that the unit and integration testing might be done iteratively, with acceptance testing occurring entirely on the final iteration of the development cycle.

By considering how the prototyping process closely follows Boehm's spiral process model, Shimeall noted parallels that led to a spiral iterative test planning process. Figure 7 illustrates this process.

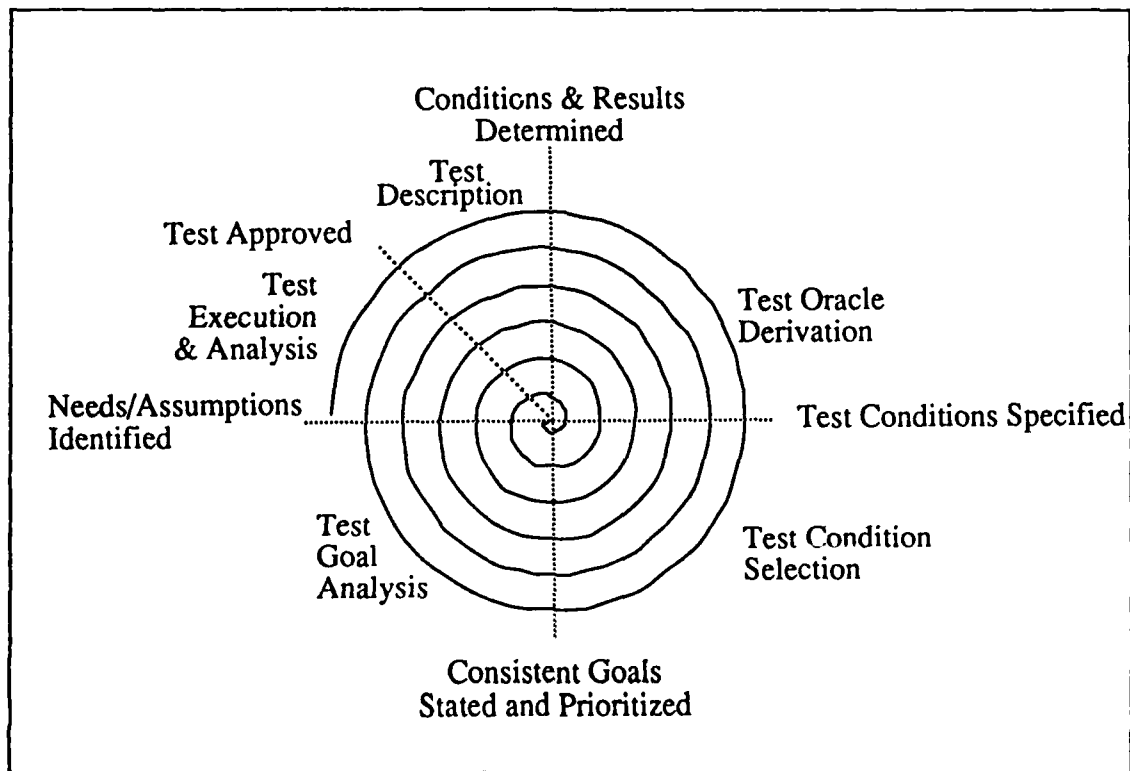


Figure 7. Shimeall's Iterative Test Planning Process [Shimeall190 p. 6]

Compare the above figure with Figure 1 from Chapter I. There is one test planning iteration for each development iteration. The opposite spiral directions in each figure indicate that the results of each stage of the cycle are being evaluated against the results of the previous stage. Shimeall reduces the issue of iterative test planning to two concerns: how to construct the initial test plan and how to build successive test plans from the initial test plan.

Initial test plans would be constructed within the existing prototyping language structure, with the test planning process beginning

by identifying the process paths that map input to output. The paths would be prioritized by characterizing the semantics of each path. By analyzing each path, data and expected responses would be identified to test typical cases, special cases, pathological cases, and illegal cases along the path. The requirements analyst (or prototyper) would be consulted in any case where specifying responses raises a problem. Shimeall observes that the requirements analysis will likely be the most time-intensive portion of the test planning process and needs automation.

Once cases were established, then the implementation modules along each path would be analyzed to select data to cause the paths to execute in the desired manner for testing. Particular aspects of each module would need to be considered, to include: similarities of the actual model's semantics to its specified semantics, differences of the module from its specification, and how compatible the module is with environmental assumptions. Analysis would likely need to be done by hand, as a general matching of performance to specifications is undecidable. Data is then selected to exercise the modules and a test oracle is derived by applying the specification to the test data to determine the expected results. Automated support for data selection and test oracle derivation is possible and desirable.

Shimeall states that for every prototype after the initial one, the testers expand the initial test plan to cover the prototype's enhanced functionality. As mentioned earlier, an iteration can change in one of three ways: changes to current specifications, additions to existing specifications, and deletions of existing specifications. In each case, anywhere that a change potentially interacts with previously tested

components, those components must be retested to see if the changes introduced any bugs.

2. Testing-related Features Needed in a Prototyping System

a. Requirements-capturing Features

Prototyping systems and their associated development techniques must capture all system requirements. One key prototyping system weakness noted throughout the thesis is that prototyping languages are geared to prototype construction and not to validation and verification. The prototypes tend to capture what is needed and not why it is needed. This is not a problem as far as construction is concerned, since anything that does not appear can be explicitly added later. The problem is that the design decisions and assumptions made during construction are not usually captured, so testers have no way to check to see if the assumptions are accurate. Prototyping systems must contain prototyping language constructs or tools explicitly to capture pertinent system assumptions and requirements. PSDL intends to provide this capability via the "informal description," and "formal description" ("axiom") grammar constructs. Implementation research is ongoing.

b. Reusable Components Features

Modules in the reusable components library should conform to a rigid coding style and documentation standard. Each module should be rigorously unit tested before incorporation into the components library. The test history should be recorded and linked to the module for reference so that during test review, testers can determine needed test data to test the module in a particular context. Unreachable paths, input/output value

ranges and such can be compared with previous test data to avoid redundant testing and prioritize additional required unit testing, if any, and prioritize testing various paths. This can save considerable required testing effort, at the price of some additional test history analysis.

The environment should maintain a module-use history so that, should a module fail in a given system, other systems using that module can be checked to decide their susceptibility to a similar flaw. The failure would be documented and possibly the module would be removed or modified to correct the flaw. In the event of correction, the modified module would require the same rigorous verification for admission into the components library. The beauty of this approach is that systems using a module continue the system verification process by flagging potential faults in other systems. By tracking module use, we reduce the risk to all consumers because, when one experiences a failure, the other systems using the module can be checked. Therefore any "sleeper" bugs not yet activated in a system are more likely to be caught before they can do damage.

c. Prototyping Language Features

Prototyping language structure should lend itself to structural testing techniques such as path testing. The implementation language also should be amenable to such testing. Thus, the reusable components can be path tested and the results stored in the testing history of the component. Also, upon implementing larger portions of the prototype, the testers can commence path testing on critical paths in the prototype's completed portions, using the structure implicit in the

prototype and its implementation language. PSDL and Ada in the CAPS lend themselves readily to structural testing. The PSDL timing constraints are examples of how language structures can provide important testing information for selecting test data to exercise modules. Techniques such as these will considerably enhance developed prototype reliability.

3. Spiral Testing: A Testing Methodology to Support Evolutionary Iterative Rapid Prototyping

The testing method proposed in this thesis follows from that of Shimeall and elaborates on his proposal with several modifications. No complete prototype environment exists yet and this forces some prototype and testing discussion to be anticipatory. For example, the CAPS probably will not have a working database until 1992-93 and the complete CAPS environment will not be available for some time beyond that. Our stated long-term goal is two-fold: to formalize the testing of rapid prototype systems by ensuring that the information needed to construct an objective basis for evaluating rapid prototype systems is captured, and secondly, to automate, or aid by automation, as much of the testing process as possible.

The proposed prototype testing methodology, termed "spiral testing," remains iterative and parallels the prototype development process. Spiral testing expands Shimeall's work by characterizing the varying types of prototype iterations and by tailoring the testing process to account for these differences. Spiral testing distinguishes between the initial few prototype development/testing iterations, subsequent iterations, and the final few iterations. The major distinction between

the first few testing iterations and the subsequent ones is the first iterations, for any but the smallest of systems, probably will have only test planning and structuring activities that establish priorities and areas of test emphasis. The framework for intermediate testing activity and final acceptance testing, to include test oracle derivation, is laid in the initial iterations. Test oracle derivation and unit and integration testing will likely be confined to subsequent and final testing iterations. Subsequent testing iterations will have less framework-related activity and more acceptance test oracle derivation activity. The major distinction between the subsequent and final iterations is that the final iterations are where developers return to their prototype to fix identified errors and testers conduct final integration and acceptance and regression testing. Figure 8 shows the separation of the groups of iterations for either the development or testing spiral. The following sections cover spiral testing in detail.

a. The First Test Planning Iterations

The first few iterations of prototype development serve varying purposes, depending on the particular software under design. When feasibility is not a consideration or when a detailed requirements document exists, the first development iterations establish the product's design framework as a base upon which to prototype the remainder of the system. When feasibility must be investigated and/or when requirements are unknown, the first several development iterations seek to construct abstract prototypes to see if an acceptable system can be designed. If the prototype is feasible, developers establish the major software

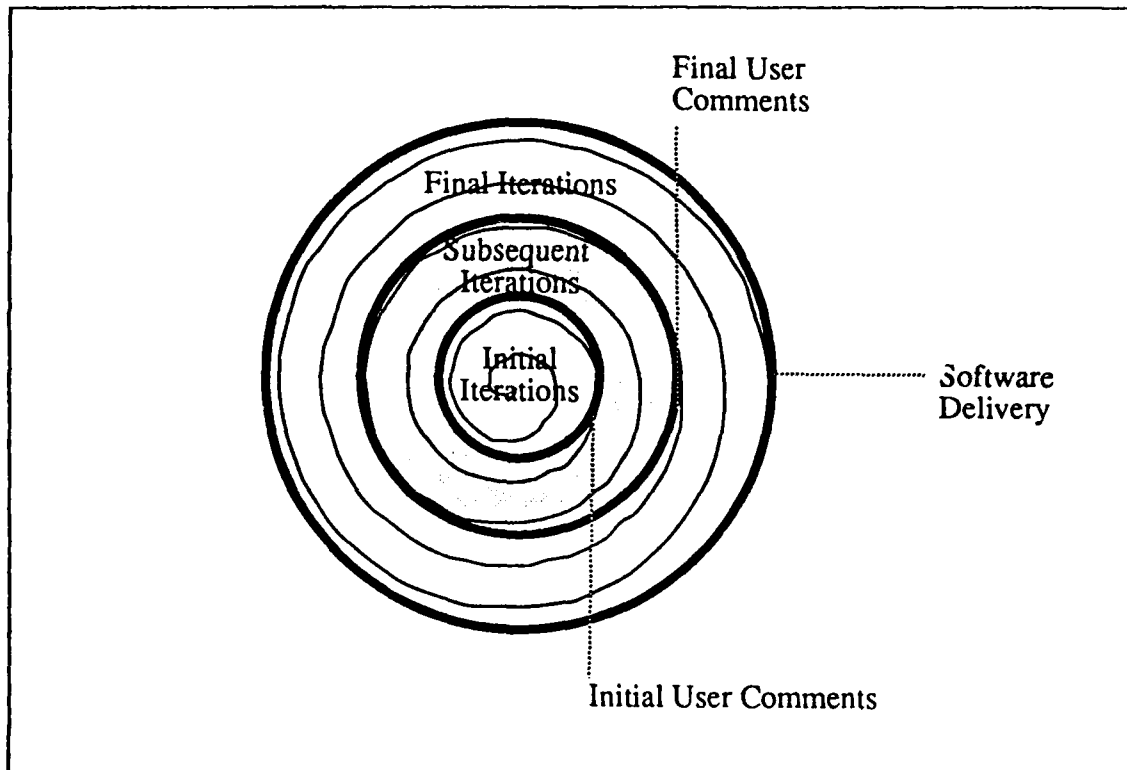


Figure 8. A "Targeted" Spiral

requirements and design a development plan upon which to build the system during successive iterations, as in the first case above. The first few development iterations will usually be devoted to establishing an overall prototype structural framework.

To mirror this process for test planning purposes, the initial test planning iterations consider the prototyping results and frame the testing process for the remainder of the project. This is the logical point for testers to determine the major critical portions of the prototype, and establish test priorities. As prototypers establish major requirements, the test team begins to break these down into test goals, determining derived goals as well. As development continues, the testers

can define the testing emphasis in greater detail, make needed test plan adjustments and record test justifications.

It appears prudent (though not essential) throughout the prototyping process for the test team to review user input and generate goals independently from the prototype team, so as to identify miss-stated requirements and to find missing requirements. This increases the quality of the prototype versions, decreasing the number of iterations needed.

The initial iterations are where the test team will forecast the most important portions of the system to test. As the implementation hierarchy of the system takes shape, the testers establish test sections for path and integration testing. The long-term testing purpose is to build the framework for constructing the final acceptance-test oracle and to fit the intermediate testing activities into the overall development plan. The process will be manual for the most part and this would be where initial testing tools and their databases/instrumentation would be initialized. The initial iterations phase would end at the prototype iteration in which the top level requirements specification is established.

b. Subsequent Test Planning Iterations

Once the basic prototype developmental framework is established, subsequent iterations commence in which developers enhance the prototype's functionality and demonstrate it for user/designer review. In the typical case, additional requirements are identified and the design matures in parallel over multiple iterations. Both are validated in the

review process. At some point sufficient requirements are identified to establish the overall system design.

The subsequent testing iterations will be characterized by unit testing, integration testing and continued requirements review to determine if there are missing requirements. To complement the design process, the test team concurrently develops integration test plans as designers complete subsections of the system design. In addition, as reusable components are instantiated to provide required functionality, the test team or design team unit tests these modules (both approaches are acceptable in current practice) by consulting the test history of the instantiated modules and conducting additional unit testing appropriate to the developing system. Should this additional testing be necessary, the test team updates the test history to reflect the additional testing and its results. This update could be as simple as appending a test script (though this could eventually cost a large amount of storage for a large components test library) or as complex as completely revising the test history to incorporate new test sets, assumptions tested, test case justifications, and additional test history details. The complex update may be needed to reduce a series of simple updates into a consistent and usefully compact synthesis. As performance aberrations are found during a given iteration's tests, they are readdressed to the design team for correction prior to the next iteration demonstration.

As the design team prototypes system components, the test team can commence integration test planning for those components. Prototyping languages such as PSDL generally provide the framework upon which prototyping tools work to select components and build interfaces for

implementation. As the components are selected and combined, the test team can begin to design their integration tests. Once a system subsection is complete, and validated by the user, the design team completes the integration test plan and conducts the test (dependent on the subsection's finality - a judgement call), providing results to the design team for correction and to themselves to guide future testing, including correction of problems in the test execution. Designers make corrections and, depending on noted problem severity, the iteration is either retested without further enhancement or else tested by appending tests to the next iteration's test plan.

The integration testing process is the same at any hierarchical system level for integration testing. The test team needs to keep integration testing at various levels coordinated to maximize test efficiency. If a standard structure for integration test sets could be constructed, then it might be possible to develop tools to manipulate these to conduct increasing levels of integration testing as more components and system subsections are implemented. Currently most of this process would be manual. Final integration testing cannot commence until the prototype implementation is complete.

Shimeall [Shimeall90: p.7] notes that it is likely that a language such as PSDL can be used to develop an initial test plan based upon the prototype language implementation. However such a plan should not be executed until all source code implementing the prototyping language specification has been instantiated. The test team can select test cases and data in advance, but as Weyuker's general-multiple-change test adequacy axiom [Weyuker88] states, two programs of the same shape

(syntactically similar) usually require different test sets. Therefore to run a test using the prototype specification for execution is not to test the implementation code, even though the specification describes the implementation. A related concern is the closeness of the match a prototyping system provides between the prototyping language specification and the selected implementation module. This match should be closely checked to ensure achievement of the desired result. Test data should be carefully selected to check the match and to ensure that any assumptions made in the module development do not conflict with the system's intended performance or operating environment.

Throughout testing, testers consult the prototype specification and all requirements to determine the correct responses to test data. Considerable information needed for test data selection will likely be found in the prototype specification language. Automated support to extract this information would be very helpful, but will depend on the prototyping language in question and in possessing the capability automatically to relate the criteria to selected test data and execute the test.

Throughout the iterations, the test methodology must remain responsive to change. Existing components and specifications may change or disappear between iterations, contingent with user/developer/tester input. Additionally, each new iteration will add increased functionality or further the completion of existing incomplete implementations. The test development process must capture all effects of change because additional testing or retesting of previously tested code may be required.

This retesting raises the issue of "phase agreement" between the prototype development spiral and the test planning spiral.

An "in-phase" agreement would have formal iteration testing proceed at the completion of a development iteration and prior to iteration demonstration to the user. The advantage here is that the test team will have tested the system and the developers are not as likely to demonstrate a system that contains undiscovered or obvious bugs. Any problems encountered in testing are corrected prior to user review. User confidence is not enhanced when bugs are discovered during the demonstration that are not related to design issues. On the other hand, many iterations will usually be demonstrating requirements not yet validated and it is wasteful to test unvalidated requirements.

An "out-of-phase" testing approach would rely on the designers to test their prototype iteration sufficiently prior to demonstration (for their reputation, not for formal testing). The test team would conduct formal testing for an iteration at the conclusion of the user demonstration. They would modify the formal test plan, developed during the development iteration, by removing any planned testing of eliminated, changed, or superseded requirements and by adding additional testing of corrections and modifications resulting from the user's review. Test planning would proceed in tandem with iteration development, but actual testing would wait for the results of the user's review. Once the testers obtain user comments, they may assume that the stated requirements at that point represent solid requirements for the purposes of testing. This assumption continues until a requirement is explicitly or implicitly changed or eliminated. The advantages of the out-of-phase testing are a

savings in testing conducted (due to testing only reviewed requirements) and increased test responsiveness to user review. The disadvantages are the increased likelihood of missed requirements and the possibility of bugs in the demonstrated system.

c. The Final Test Planning Iterations

Once developers establish all requirements (usually in the latter iterations), the final few iterations of development are devoted to implementing the remaining functionality, followed by error correction. Therefore the testers can devote their work to completing the test oracle for acceptance testing, and to remaining unit testing and subsection integration testing.

The final test planning iterations commence with the completion of the operational prototype and prior to final user acceptance. As any final requirements are implemented or as system components are fine tuned, tests are developed and conducted to cover these changes. Most importantly, the test team completes the acceptance test plan. Once the system is completely implemented and the acceptance test design, including the test oracle, is complete, the test team conducts the acceptance test. The test team checks differences in actual results from expected results and corrects the tests as appropriate while the design team corrects system faults. The cycle continues until the system successfully completes testing. If the design team is busier than the test team then the test team can use the available time to conduct additional testing or priority-superseded testing previously skipped. The result should be a sufficiently tested software system.

d. Spiral Testing: Advantages and Disadvantages

Spiral testing has the advantages of being flexible and maximizing the amount of testing conducted during prototype development. It allows for the steady development of an acceptance test in the face of continual system change and facilitates lower level testing as soon as implementation code is instantiated. The spiral testing approach particularly suits the methodology for use with evolutionary iterative rapid prototyping that is itself spiral. Using test histories for reusable components should speed the testing process by reducing the amount of unit testing required. A further benefit of the test history feature is the compounding of unit testing for reusable components with use, either increasing our component confidence factor or at least delineating the bounds in which it may be successfully used.

The spiral testing approach also results in thorough documentation of the testing conducted and in a formal, written test plan that can be reviewed with the user for approval. The extended time for test development (considerably more than in conventional life cycle models) also should provide a more thorough test.

The major disadvantage to the approach is that the final acceptance test remains a moving target until the completion of implementation coding. Additionally, the test team must remain vigilant against following the development process so closely that they fail to view the system objectively from the outside. The first disadvantage is inherent to the prototyping process, therefore our goal is to minimize its effect. Spiral testing is likely to do this. The second disadvantage may

be reduced with experience, but will likely require separate test teams to conduct critical acceptance tests.

One should note that the methodology remains a theory at this point. Further research will be required to determine its feasibility and practicality. Refinements and amplification will almost certainly be required. Chapter V addresses recommendations and conclusions regarding spiral testing.

e. Characteristics of Candidate Software Testing Tools

The amount of detail present in the evolution of software requires tools that could correctly capture information from prototype iterations. This information comes from sources including both the prototyping specification language and the implementation language. Once captured, the information must be suitable for review. For testing we must automate assumption recording in the design process and explicitly capture requirements, whether we list these requirements in text form or else have a sufficiently powerful and expressive prototype specification language to cause the requirements to be recoverable. The latter sounds appealing but does not appear to be feasible in the near or short term. These tools also must support a huge amount of information gathering and review and be fully integrated with the remainder of the prototyping environment.

The tool integration issue is essential in large systems to ensure consistency maintainance between all aspects of the prototype and the related software life cycle. The requirements, specification, reusable components, design histories, version control, test cases, test

histories, test oracles and such must remain as consistent as possible, without corruption or else the developers lose the ability to assure correct implementation and sufficient testing. Testing tool integration must preserve the security of the test team from the development team. Some argue that one value of a test plan is that the designers plan to ensure that the implementation handles all the test cases. There is value in this visibility but a concern is that if tests become corrupted, then they will not accomplish their intended purpose. Access to testing tools, at least the set used by the test team, should be controlled.

Testing tools must allow the testers to predict important test cases and strategies based on a knowledge of the prototype. The tools must respond correctly to change between iterations. Because of the size of most modern software systems, the tools should help the testers to aggregate, prioritize and record test cases, providing the test team with feedback on coverage of code (all statements, all paths, etc.), and range of possible test cases (typical, boundary, pathological, load). Many tools are possible now and more will become so as prototyping research matures. The challenge now is to hone in on those testing tools that are currently feasible and will provide us with the most assistance for the effort expended.

D. REQUIREMENTS-BASED TESTING: A KEY METHODOLOGY FOR RAPID PROTOTYPING

The key system developmental concerns in rapid prototyping are to determine, demonstrate, and meet the user's requirements. At the center of all this are the system requirements. To test a prototype at all, testers must know objectively what it is to do. The testers must capture

requirements explicitly. Requirements must be restated as test goals so that they are objectively testable and reflect a single behavior for which we can ask, "Is this behavior or characteristic present?" The primary rapid prototype testing method will therefore be requirements-based. This is not to say structural testing is not needed. Both are necessary, but structure implements function and does not define it in a way readily apparent to humans. Requirements-based testing puts the desired program characteristic directly before the tester and begs validation. Since prototyping is most likely to be used in situations where requirements are not well known, requirements-based testing will particularly complement the prototyping methodology.

As early as 1982, Taylor and Standish noted that prototypes are used to determine if requirements are adequate and that a way was needed to perform this task systematically [Taylor&Standish82]. They then go on to say that testing needs to be coupled with requirements review and that a "systematic traceability of the requirements to the tests that elicit the behaviors that are supposed to satisfy them seems called for." This seems straight-forward enough and may well be further enhanced if one also can couple this with a mapping to the tested system's implementation code.

The process of requirements-based testing was described in Chapter II. The first step in aiding testing within evolutionary iterative rapid-prototyping systems is to develop a tool to aid in requirements-based testing. The Test Goal Tracking System (TGTS, pronounced "Targets") provides testers with a way to capture explicitly every stated or derived requirement for prototypes developed with CAPS. It further allows the tester to record pertinent test information concerning each requirement,

such as test priority, test class, test aggregate, iteration in which requirements were added (and deleted), and a test history for that requirement. Each requirement maps directly to the portion of the prototype that implements it, so that the requirement can be traced and allows requirements to be cross referenced to determine where one requirement's implementation may interact with other requirement's implementation.

Shimeall [Shimeall90: p. 8] highlighted the need for tools like TGTS by noting that the starting point for automating prototype testing is to build tools to capture information on the intent of change to ascertain if the intent was achieved. Since most prototyping languages are geared to system construction and not to validation and verification, we need tools that will augment current prototyping systems to allow us to capture requirements and intentions and assumptions. TGTS is designed to do this.

A chief value of TGTS is that it helps testers deal with change in the prototype. When part of an implementation changes, TGTS can be used to show testers what portions of the implementation have changed and what other requirements may be affected. Further, since TGTS also cross references into test aggregates, classes and priorities, testers also can work to maintain consistency and completeness in the test plan. Thus testers will know what portions of test plans may need revision or retesting to ensure adequate test coverage. All the cross referencing is automatic, thus saving testers considerable effort.

TGTS is designed specifically for spiral testing and is intended as the first in a series of companion testing tools for CAPS. It is a menu-driven tool, that will work well in a window environment with CAPS.

Chapter IV provides a detailed description of TGTS and a small example of its use in a system under development.

IV. TGTS: A SAMPLE REQUIREMENTS-BASED TESTING TOOL

A. AN OVERVIEW OF THE TEST GOAL TRACKING SYSTEM

The Test Goal Tracking System, or TGTS, helps testers explicitly capture test goals for prototypes developed with the CAPS. Testers derive test goals from system design requirements and assumptions, restating them for testing purposes. TGTS is a database tool that maps test goals to their source in PSDL/source code for cross-referencing and retrieval. It allows testers to track information about a test plan's test goal coverage (how, where, why), and about test goals that do not yet appear in a prototype.

1. TGTS Distinction

Some may initially be inclined to levy the criticism that prototyping captures the system requirements in the prototyping process, so why bother recording requirements separately? Isn't TGTS just duplicating effort? The response is, "No" simply because what TGTS does is record explicitly what already occurs implicitly in the prototyping process. One cannot test implicit requirements with any certainty of the completeness or sufficiency of the test. Testers are commonly concerned with sufficient testing, particularly in the embedded, real-time programming environment that CAPS handles. TGTS does not add a redundant process to prototyping, but simply helps testers capture results of what is already occurring in the prototyping process, namely requirements verification and validation. All this is to say "You can't test what you

don't know and you can't know the requirements unless you explicitly capture them, particularly for complex systems." Therefore, TGTS allows the system requirements captured and/or validated by the prototyping process to be developed into test goals and recorded for testing purposes, mapping each to the implementing code in the prototype.

One of prototyping's primary purposes is requirements-capturing. If one expends effort to prototype and determine or validate requirements, then one should capture the requirements as well as the prototype. It is inefficient to fail to capture explicitly requirements (since it may be accomplished in parallel with the construction process) and then turn around and have the testing process grind to a halt while testers scramble to figure out what behaviors they are trying to verify. In the end, prototyping will be slower for evolutionary systems unless testers readily capture requirements and establish concise, clear and complete test goals.

2. TGTS Goals

The primary goal was to provide a tool to map captured and/or validated requirements to the prototype. The tool needed to capture accompanying test-related information (such as test priority, aggregation, and classification) to aid test planning and execution. The tool also had to be flexible enough to capture change in requirements between iterations and maintain consistent mappings to concurrently changing implementations in CAPS.

A second goal was to show the spiral testing methodology of in Chapter III to be a non-intrusive testing approach within evolutionary iterative rapid prototyping. Further, the spiral testing approach needed

to be shown to be appropriate for test tool design supporting rapid prototyping.

Testers may record a history for each test goal, describing how that test goal has been met or noting what remains to be tested. They also record current status, including the iteration in which they added the test goal and the iteration in which they last modified the test goal's history.

Concurrently, TGTS allows testers to track the status of the PSDL prototype, noting the implemented operators and data streams and their degree of implementation. Testers record the iteration in which the various operators and data streams are added and fill in the accompanying history for each. By combining the test goal and PSDL information, testers can more readily decide for a given iteration what should be tested and how.

3. TGTS Database Software

The database software chosen as a basis for TGTS was dBase III+. The primary reasons for this choice were: software availability, software familiarity (thus reducing the learning curve), dBases' rapid database construction facilities, and the availability of PC's and workstations to run TGTS. Additionally, many computer users are familiar with dBase, making further research more readily accessible to them.

As with any design decision, trade-offs exist. Certain weaknesses resulting from using dBase III+ include: limited data types and allowable manipulations, lack of a full SQL approach to relational databases, and limitations inherent with certain data field types in

dBase, such as memo fields. Additionally, unless TGTS is compiled with a dBase compiler, such as Clipper, it only will run inside dBase. Since dBase III+ is a DOS operating system application, TGTS can only be used on workstations that execute or emulate DOS. Nevertheless, the tool works quickly and meets current research needs. It is also simple and flexible enough that new users can quickly become familiar with it.

B. DEVELOPMENT AND DESIGN OF THE TEST GOAL TRACKING SYSTEM

1. Detailed Design Decisions and Tool Structure for TGTS

The TGTS design philosophy was to build a simple, user-friendly, tool with a very direct modular implementation to be used, studied, and evaluated over time. The tool design needed to be easily modifiable and extensible over time, as experience provided sufficient input to determine what operations, utilities, and output would best suit CAPS prototype test development.

a. Modular Design

TGTS has a modular structure so that the system may be easily modified. Module coupling is minimal in the program, other than the menu modules linking to the called application modules. Minimal coupling makes future tool modifications easier to effect. Should users desire added functionality, the tool can be extended by simply adding additional program modules and then including appropriate calls to invoke the new module(s). The modular approach is also the programming style around which dBase programming is designed. The dBase program execution is a series of module calls, depending on the functionality being invoked. The conditional program calls are effectively and easily implemented with an

interactive, menu-driven program approach. The following figure shows the top level program decomposition. Note that the decomposition groups

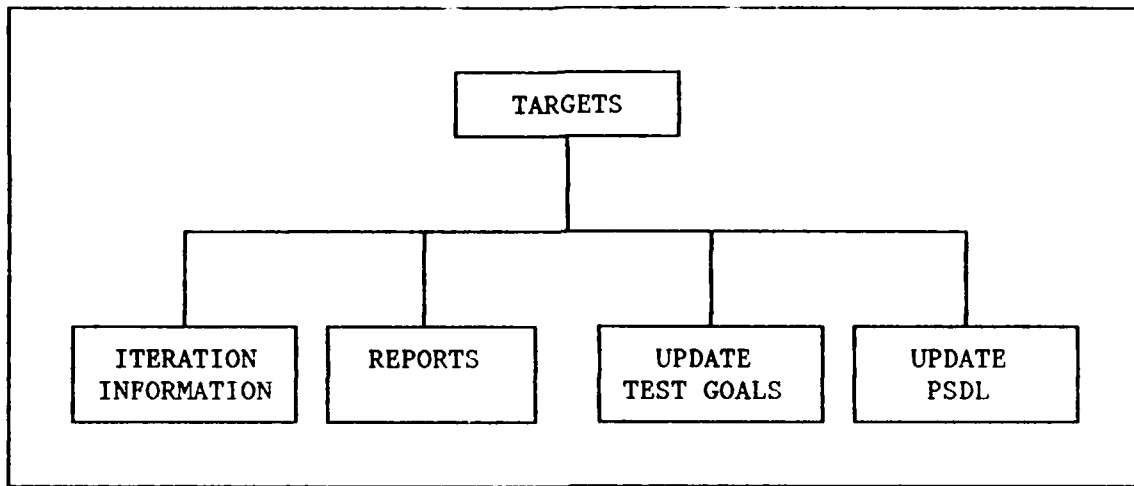


Figure 9. Top Level TGTS Program Decomposition

logically related tool functions together. For example, the "Reports" module contains all the output functions for TGTS and the "Update Test Goals" module contains the functions that manipulate the test goals database.

b. Menu-driven Format

The TGTS program is menu-driven, both allowing users to see TGTS functionality options immediately, and keeping the tool simple and easy to use. The tool's functionality is "set" in the sense that user-performed operations only include those explicitly accessible from the menus. Of course the serious dBase III+ programmer has considerable latitude to manipulate TGTS directly from the dBase command line, apart from the TGTS program. While not recommended for typical usage, command line tool/database modifications can actually benefit development and

research because it allows the test databases and the programs to be modified quickly, as research might dictate.

c. Tool Input and Output

Standard input for TGTS is via the keyboard and the database and index files contained on disk. Standard output for TGTS may be optionally directed to the screen only or the screen and standard system printer. DBase III+ automatically updates database and index files to reflect changes in data contents. The user navigates menus by choosing the number corresponding to the desired action. Data fields requiring input have accompanying guiding statements, where appropriate. TGTS guards data input fields to accept input only within designated ranges. DBase III+ provides screen entry programming facilities to conduct data input checking that ensure data is of the appropriate type and range.

d. TGTS Database Design

TGTS is a multiple database application, using three main databases. The first database (GOALS.DBF) is a test goals database used for recording and referencing all prototype test goals. The second database (OPSTREAM.DBF) is the PSDL operator and data stream database, used for recording and referencing information on all the PSDL operators and data streams. Finally, the link database (RO_LINK.DBF) contains data records linking test goals to operators and/or data streams that implement the prototype system requirements (restated in test goal format). The mappings between the first and second database may be one-to-many, many-to-one and many-to-many. Usually they will be many-to-many. The following figure illustrates the relational structure of the database.

The iteration information database (ITERATNS.DBF) is not linked to the other three databases and is used to record overviews of each iteration's history.

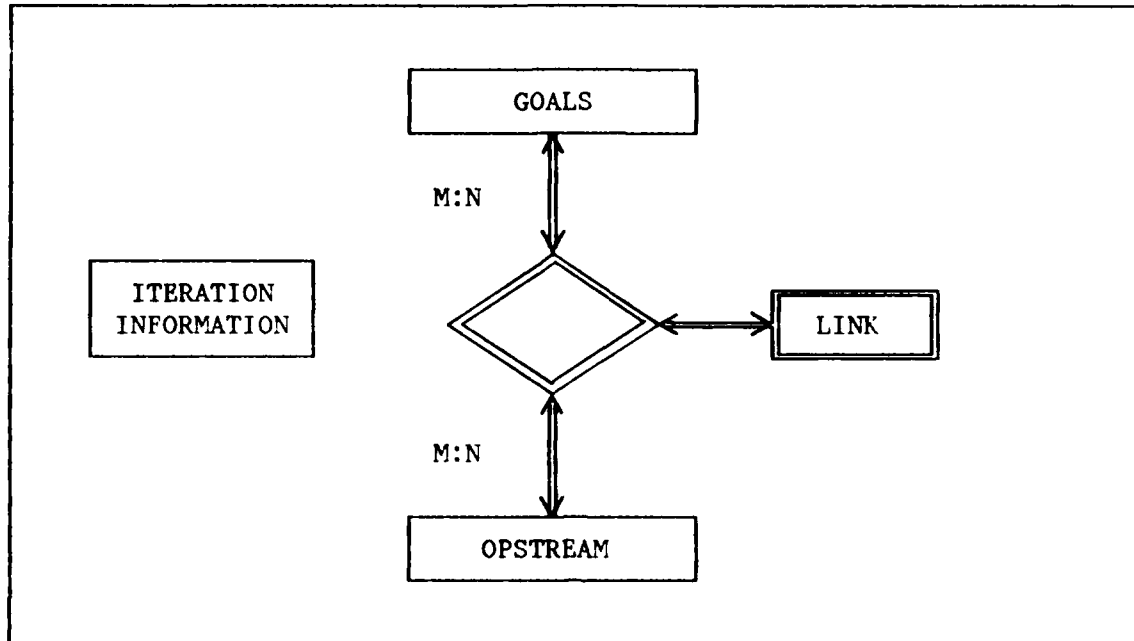


Figure 10. TGTS Database Structure

The database fields are excluded from the above figure to avoid clutter. The following table lists the fields and field characteristics of the database records and includes the dBase III+ field types and index files relating to fields. DBase keeps the index files listed to index records. Each index file's key is the respective field name beside which the index file is listed. Each of the first two databases assigns a unique number to each record that serves as a key for referencing the record. This allows the link database to establish unique links simply by referencing the two numbers. The field names in Table 1, are written out descriptively. The field names in the source code are

TABLE 1

GOALS.DBF

Field	Field Name	Type	Width	Range	Index File Name
1*	Goal Number	Numeric	6	1-9999	GOAL_NUM.NDX
2	Goal Description	Memo			
3	Test Priority	Numeric	1	0-4	TEST_PRI.NDX
4	Aggregate	Numeric	2	0-99	AGGREGAT.NDX
5	Updated	Numeric	2	0-99	UPDATED.NDX
6	Iteration Added	Numeric	2	1-99	ITER_ADD.NDX
7	Test Class	Numeric	1	0-4	TESTCLAS.NDX
8	Goal History	Memo			
9	Deleted	Numeric	2	0-99	DELET_TG.NDX

OPSTREAM.DBF

Field	Field Name	Type	Width	Range	Index File Name
1*	Operator Number	Numeric	6	1-9999	OP_NUM.NDX
2	Operator Name	Character	40		OP_NAME.NDX
3	Operator	Logical	1	T or F	
4	Updated	Numeric	2	1-99	O_UPDATED.NDX
5	O Iteration Added	Numeric	2	1-99	O_ITERAD.NDX
6	Operator History	Memo			
7	Deleted	Numeric	2	0-99	DELET_OP.NDX

RO LINK.DBF

Field	Field Name	Type	Width	Range	Index File Name
1*	Goal Number	Numeric	6	1-9999	GNUMLINK.NDX
2*	Operator Number	Numeric	6	1-9999	ONUMLINK.NDX
3	PSDL Part	Character	40		PARTLINK.NDX
4	L Iteration Added	Numeric	2	1-99	L_ITERAD.NDX
5	Deleted	Numeric	2	0-99	DELET_LK.NDX

* - INDICATES KEY FIELD

shorter to fit dBase programming requirements. The ranges chosen for the fields are arbitrary and provide sufficient range for anticipated projects.

Several field data ranges limit input, either to restrict the possibility of errors or because each possible value represents something. For example, there are four test priorities and four test classes

possible, with an entry of zero signifying that testers have not yet assigned priority or class. Each number allowed in the "class" field represents a test class: 1 - non-testable, 2 - inspection, 3 - analysis, 4 - execution. The TGTS User's Manual, in Appendix B, describes all the database fields in detail.

2. Tool Operations with TGTS

The primary purpose of TGTS is to record test plan information on a developing prototype's test goals and then map the test goals to their source in the prototype code. As TGTS' top-level module decomposition shows, the operations fall into four categories: database outputs, test goal updates, PSDL updates, and iteration information. The principal operations of each will be discussed in turn. A complete description of TGTS' operations is contained in the TGTS User's Manual, Appendix A.

a. TGTS Database Output Operations

The key to TGTS effective use is the proper presentation of the database contents. The information provided must give testers the information needed to develop a thorough test plan. Ideally, it also will aid development personnel by pointing them to likely sources of errors when tests detect faults. TGTS provides these services with sixteen output operations, each implemented as a separate module.

For each output operation, the user has the option to receive the output on the screen only or else on the screen and printer. Once projects grow to any sizeable extent, testers will usually need printed output. TGTS groups output listings as either "Test Goal Output Options," "Operator/Data Stream Output Options," or "Link Output Options." The

"Test Goal" and "Operator/Data Stream" options further allow the user to specify either a brief or descriptive output of the database records. The brief description excludes the descriptive text fields for each record, while the descriptive output includes these text fields.

At the most basic output level are the "All" output options. A user may list all contents for either of the three databases, which is useful in many general ways. In each of these listings, the lists may be indexed by any of several keys to provide users with appropriate record groupings.

More specific listing operations exist. Test goals added, changed or deleted in a given iteration as well as those of a given test class or priority or aggregate may be listed. In similar fashion, operators or data streams added, changed, or deleted in a given iteration may be listed. Particularly helpful are the listings that show all the operators and data streams that map to a particular test goal, and its opposite, the listing that shows all the test goals that map to a particular PSDL operator or data stream. The last two listings are especially helpful in test planning because they let the testers rapidly update tests to account for implementation changes and to track implementation code test coverage. When developers change an implementation module, the testers can see which test goals are most likely affected. When testers change or delete a test goal, then they can note appropriate test modules for additional testing consideration.

Finally, an output operation exists to list all the links between test goals and PSDL parts added in a given iteration. This helps provide a history of test development concurrent with the system

development process and serves as a test plan management aid. When testers combine this last listing with detailed listings of the other databases, they have an overview of the entire requirements-based test plan.

b. TGTS Test Goal Update Operations

Testers derive requirements-based test goals from the validated system requirements for a given software project. Previous research has described various sources for these requirements [Hernandez89]. Once the testers develop the requirements into test goals, the test goals are added to the database, annotated as appropriate and mapped to the prototype implementation code.

TGTS provides five operations upon test goal database records. These are very straight-forward. First, a test goal may be added. Initially, testers fill the test goal record's description field, followed in the goal history field by a justification of the goal's derivation, such as "user comment on iteration x demonstration" or "initial requirements document, requirement no. 23" or some such sufficiently clear description so that the test team knows explicitly where the goal originated. The "Add" operation automatically assigns test goal numbers sequentially. This prevents two test goals from having identical test goal numbers and also allows the numerical value of test goals to reflect the sequence of test goal derivation, should the test team deem such information valuable.

The second operation is the "Delete" operation. This operation marks a test goal as deleted from the test plan. The test

goal's record remains a part of the test goal database with a deleted field flagged. This operation is particularly valuable because it allows test goal information to be retained for record purposes and keeps records that may have to be added back in a subsequent iteration, should the results of an iteration demonstration dictate. Testers may review a given iteration's deleted test goals to determine what test plan modifications will be required in light of requirements changes. This operation is a bit more complex than previous ones in that it requires TGTS to delete all link records that mapped to the deleted test goal. Note though that the links are non-destructively deleted in the same way as the test goals so that they may be added back, if necessary.

The third operation is the "Change" operation. This operation does not allow testers to change the nature and description of a given test goal per se. The "Change" operation allows testers to change test plan related information concerning a given test goal, such as its test priority or test aggregate or it allows testers to fix incorrectly added information in a given test goal record. The reason for this approach is basic. A test goal itself does not change. It is either a valid test goal or it is not. If it is invalid, then it should be deleted. If it is valid, then its description remains unchanged. A brief digression to explain the meaning of an inappropriate test goal is appropriate.

A test goal may be inappropriate because of many reasons. First, it may simply have missed the mark of system requirements completely. Second, a test goal may be invalid because it was not specific enough, combining too much into one test goal. Testers may then

delete the initial test goal and replace it with multiple succinct test goals (here, justification comments referring to the initial test goal may be appropriate for test history purposes). Third, a test goal may not be feasible in its current state or may require modification to conform better to the overall test plan. Again, justification comments would be appropriate.

The "Change" operation has an additional important function, namely to undelete previously deleted test goals. Should a previously deleted test goal need to be reinstated, then the "Change" operation undeletes the test goal by marking the record's deleted flag as undeleted. Further, it then allows the user to undelete all the test goal's previous links or else selectively revive test goal links to account for changes that may have occurred in the prototype's implementation code.

Test goals must be linked to the prototype implementation code, once developers add the functionality to the prototype. The "Link" operation links a test goal to an existing operator/data stream record. The "Link" module implementing this operation also works in the opposite direction, linking operator/data stream records to existing test goal records, making it one of TGTS' few modules that is called by multiple modules. The "Link" operation explicitly allows the testers to link test goals to the prototype implementation code.

The final operation is the "Unlink" operation. "Unlink" allows links to be removed between a test goal record and an operator/data stream record. This allows the link database to be manipulated to account for changes in prototype implementation. Like the "Link" operation, the

"Unlink" implementation is a shared operation with the operator/data stream database.

c. **TGTS PSDL Update Operations**

The PSDL database is subject to five operations similar to the test goal database operations. Two of the operations, "Link" and "Unlink," are identical to those described above. The other three, "Add," "Delete" and "Change" are similar to the analogous operations for the test goals database, but have input screens to manipulate operator/data stream database records.

The "Add" operation, like its counterpart allows implemented PSDL operators and data streams to be added to the database. There is an "Operator" boolean field in each record to flag each as either an operator or a data stream record since both record types reside in the same database. Each record receives a unique operator number to distinguish each part and provide a key for linking PSDL parts with test goals.

The second operation is the "Delete" operation which non-destructively removes a PSDL part record from the database. Again, should a future change reinstate the PSDL part, it can be retrieved from the database. When a user invokes the "Delete" operation, the links related to the record are also non-destructively deleted.

Lastly, the "Change" operation serves the same purpose for PSDL parts records as its test goal namesake operation. Historical information can be updated and previously deleted PSDL parts records can be undeleted. As before, the associated links can be batch or individually undeleted.

d. TGT'S' Iteration Information Facility

TGT'S has a fourth database that provides a descriptive overview of iteration information regarding a project's design and test goal development. While this database is not directly related to the main three databases, it allows test personnel to record information about an iteration's development deemed important for testing purposes. Such information might include: major areas of functionality added in a given iteration, key user responses to a given iteration demonstration that bear on testing, and major portions of the test plan for a given iteration.

The iteration information's design is very basic consisting of two fields: the iteration number field and the iteration history field (a descriptive text field). Operations allowed on this database include adding, deleting or changing an iteration record, listing iteration information for a given iteration and listing the last iteration number used. The iteration information database can be used in whatever way the test team decides is most beneficial for a project's needs. Since the main information field is a text field and dBase does not manipulate it in any way, almost anything deemed appropriate may be recorded. Experience and/or particular project needs may dictate redesigning this database to allow a more structured form of iteration data storage.

C. USE OF THE TEST GOAL TRACKING SYSTEM

The purpose of this section is to provide an example of TGT'S usage with a software prototype. To make the example practical and relevant to current prototyping research underway at the Naval Postgraduate School, I chose a published CAPS prototype specification. This publication, A

Software Prototype of the Message Processor in Navy C3I Station, [Luqi&Davis89], provides a set of software requirements and a PSDL prototype decomposition suitable for an example. Since the specification only covers the upper levels, it is modified and extended slightly with dummy requirements and operators to allow particular TGTS capabilities to be demonstrated. Only selected portions of the prototype specification are included in the example for simplicity and brevity. The reference provides the entire specification. Finally, Appendix A provides the complete TGTS database for the example that follows.

1. Problem Environment and Prototype Specification

The C3I project entails ongoing research at the Naval Postgraduate School to provide the U.S. Navy with a low cost Navy Tactical Data System (NTDS) display using commercially available resources. Researchers saw this as an ideal application for PSDL and rapid prototyping due to the time constraints requirements for this large real-time system. A Command, Control, Communications and Intelligence (C3I) System provides commanders with an information tool to aid in seeing the tactical situation within their area of responsibility. Many processes exist within a complete system, but the present example concentrates upon a Message Processor (MP) component for routing:

- track reports,
- participating unit directives,
- status reports, and
- combat directives

between a Direction System (DS) with Onboard Sensors (OS), and various radio transceivers.

2. Message Processor Requirements

System design commences with the design team determining the system requirements. Generally, the Message Processor (MP) must offload the following data:

- message routing information,
- ship link track management,
- unit status,
- directive management, and
- communication link functions.

Given the top level system functional specifications (omitted here, see [Luqi&Davis89: p. 5]), a system can be decomposed into a top-level PSDL module decomposition. As developers validate detailed requirements, the subsequent modules to implement the requirements can be designed. Figure 11 shows the decomposition that system designers built.

For the purposes of this example, only the Manage Radar Tracks sub-module's detailed requirements will be listed. The radar track manager is responsible for processing all SL and AL tracks, which originate from other C3I modules (SL and AL). The radar track manager fulfills the following requirements:

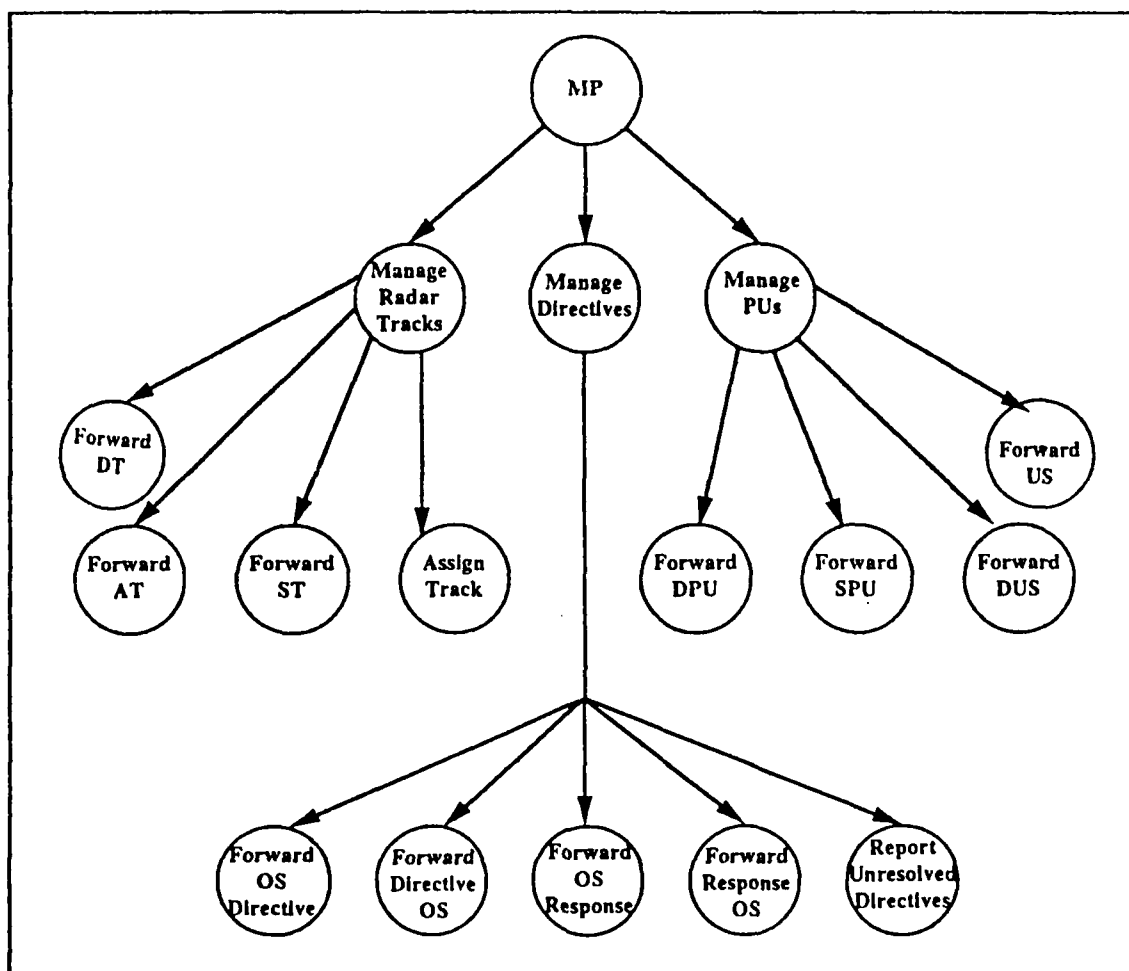


Figure 11. C3I Message Processor Module Decomposition [Luqi&Davis89: p.11]

1. A DT report from DS shall, if possible, be assigned a free or lowest priority track-id (if current id is null) and broadcast over SL within two SL cycles of its arrival at the MP.
2. Each ST that is broadcast over SL shall be routed to DS if it is consistent with the receiving ship's current SL tracks; otherwise one of the two inconsistent track reports shall be selected (to become the current track with that id) and routed to DS. The ST is analogous to a remote track in NTDS link-11 capable ships.

Amendment - Each ST that is broadcast over SL shall be routed to DS if it is consistent with the receiving ship's current SL tracks; otherwise a correlation conflict warning shall be sent to the DS. Both tracks shall be selected and routed to the DS for resolution by the system operator. The system operator

must have control over traffic conflicts. MP conflict resolution recommendations may be forwarded to the DS to assist the operator.

3. In the absence of communications failures, a concurrent allocation of the same SL track id to two inconsistent tracks by different ships shall be resolved.
4. In the absence of communications failures, a concurrent assumption of reporting responsibility for the same SL track shall be resolved.
5. A base unit MP shall route to AL those SL tracks with ids that are designated by DS (with TA messages) for reporting over AL (irrespective of which ship has reporting responsibility). This routing shall persist until a TA message retracts it or the track is dropped.
6. Each track report in a packet received from a ship or aircraft shall be processed or discarded within one link cycle of its arrival at the MP. That is, before another updated or new track can arrive from the same unit.
7. Each DT message from DS to MP shall be transmitted over SL within two link cycles of its arrival at MP unless there are more than 225 words of message data to be transmitted in the next outgoing packet; in that event queued messages shall be deleted or replaced in order of priority until the next transmission opportunity. The message handling priorities are:

- ISL, IAL, SLS, and ALS
- incoming and outgoing directives
- outgoing messages (from base units) to remote units (aircraft)
- incoming SL and AL track reports
- outgoing SL track reports
- all other messages (including responses to directives)

*Amendment - The message priorities should be amended as follows:

- ISL, IAL, SLS, and ALS
- hostile quick response tracks (such as pop-up missiles). These targets shall be identified by a special field in the track descriptor.

- incoming and outgoing directives, including responses to directives
- outgoing messages (from base units) to remote units (aircraft)
- incoming and outgoing hostile and unknown DS, SL, and AL track reports
- incoming and outgoing friendly SL and AL track reports
- all other messages.

For the purposes of the example, the abbreviations used above are inconsequential. The amendments listed will be used to show the responsiveness of TGTS to change in requirements.

From the above requirements, the designers decompose the manage radar tracks sub-module shown in Figure 12, into the PSDL description shown in Figure 13.

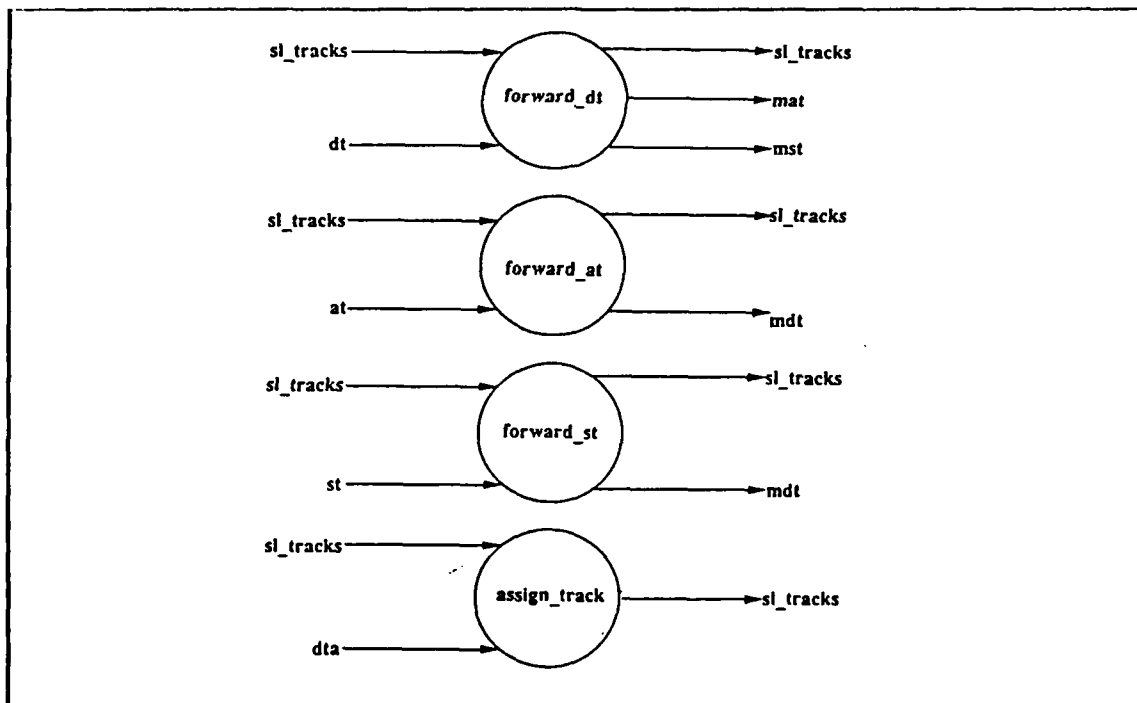


Figure 12. Manage Radar Tracks Sub-module [Luqi&Davis89: p. 14]

The textual description contained within the PSDL operators are comments that can be a rich source for test goals and should be thoroughly examined. Additionally, they can reflect assumptions made and serve to amplify designer's intent for future lower level modules.

CONTROL CONSTRAINTS

OPERATOR forward_dt MINIMUM CALLING PERIOD 20 ms
MAXIMUM RESPONSE TIME 500 ms

- worst case: each DT message from DS must be
- forwarded over the link within 2 ship link cycles
- worst case: no more than 50 DT messages/second shall
- arrive from DS to the MP
- 2 ship link cycles = 1000 ms available to forward a
- maximum of 50 DT messages, gives minimum
- calling period of (1000/50) seconds = 20 ms
- Minimum link cycle duration is 500ms, thus each module
- must be capable of completing computation within this
- time to handle worst case of a two ship link with few
- messages to forward, thus maximum response time is
- 500 ms
- allocate 500 ms to mp and 500 ms to sl
- Max latency considering both RF propagation delay
- and DMA latency is 7.25 ms for ship link and 6.2 ms
- for aircraft link

OPERATOR forward_at MINIMUM CALLING PERIOD 5.8 ms
MAXIMUM RESPONSE TIME 500 ms

- system maximum track load is 256 tracks
- radar track manager contains 3 modules responsible
- for forwarding messages over link
- worst case assumption is that this module will have
- 256/3 track messages for forwarding within one
- link cycle of 500ms, therefore this module must
- be capable of forwarding one message every (500/256)x3
- seconds = 5.8 ms, which gives the minimum calling
- period of the module
- Maximum response time as for forward_dt
- Max latency for aircraft link is 6.2 ms

OPERATOR forward_st MINIMUM CALLING PERIOD 5.8 ms
MAXIMUM RESPONSE TIME 500 ms

- Minimum response time as for forward_at
- Maximum response time as for forward_at
- Max latency for ship link is 7.25 ms

END

Figure 13. PSDL Description of Radar Track Manager [Luqi&Davis89: p. 19]

3. Test Goal Development Process with TGTS

Assume that the first development iteration produced the system decomposition given above, with the requirements listed, not to include the amendments. The test team must translate the requirements into test goals and load them into TGTS, along with the PSDL operators that define the decomposition. To the extent that requirement implementation is defined, the test goals may be linked to the PSDL implementation.

As an example decomposition, requirement one would become two test goals, each describing an individual testable behavior:

- A DT report from DS shall, if possible, be assigned a free or lowest-priority track-id (if the current id is null).
- A DT report from DS shall be broadcast over SL within two SL cycles of its arrival at the MP.

Testers simply break compound requirements into their components. The process continues until the requirements are all restated in an acceptable test goal form.

Some requirements are derived, implicit or based upon system assumptions. An example of this sort of requirement occurs in determining the acceptable link cycle duration. Without deriving the requirement in detail (done in [Luqi&Davis89]), derivation factors include the assumed maximum distance between two communicating stations, radio frequency propagation velocity and implementation architecture assumptions, all of which testers need to know and record as test goal justification. The resultant test goal is: "Minimum ship link cycle duration is 500 ms." Appendix A contains the sample database at the conclusion of the first development iteration.

After TGTS database loading, all the top-level and MP PSDL operators are contained in the operator/data stream database and the test goals are contained in the test goal database. By examining the modules, the test team can determine which links, if any, can be established. The test team establishes links for the module level at which a test goal's implementation becomes explicit. They continue to establish links at successively lower operator (module) levels that further the implementation, continuing the process to the source code (reusable component) module level. For example, if the "forward_dt" module had four sub-modules, "W," "X," "Y," and "Z," and two of them, "W" and "X" implemented the ship link cycle length (500 ms maximum - derived requirement from previous paragraph, then the "minimum ship link cycle of 500 ms" requirement would have a link to three modules: "forward_dt," "W" and "X." Each link record would contain the portion(s) of the PSDL specification to which the test goal was linked. The "forward_dt" link would state that the PSDL implementation part would be "Operators W and X" because they implement the requirement. Testers establish links only for modules that actually help implement the requirement, not just simply encompass it somewhere at a lower level. This is important to minimize the number of links and to cause the links to point to the prototype parts that effect the test goal's implementation.

As testers fill database records, they can classify test goals by how they will be evaluated (inspection, execution, etc.), fill in the iteration in which testers added the record, and such. For test goals, the test priority, test aggregate, and test history fields will only be filled as testers make those decisions and the actual test plan design

matures. Successive iterations may produce requirements changes like the requirements amendments listed previously. When this happens, the test team rewrites the appropriate test goals. Prior to inserting the new test goals, the test team uses TGTS to trace the old test goals' implementation in PSDL. The old implementation and any test plan aggregates/accompanying test oracles (if developed) are reviewed and compared with the new implementation to determine what must be modified, deleted or superceded. TGTS users add and link the new test goals and PSDL parts, and then delete the superceded ones and their links. In both cases, testers comment history fields to reflect the change justifications. Additionally, testers update the iteration information database to reflect a summary of the changes. Appendix A provides a before and after TGTS database reflecting the effect of Requirement Two's amendment and samples of Iteration Two's database contents.

Once developers complete a portion of the prototype, that portion's test development can be completed (usually, barring additional test dependencies). Testers can use TGTS output to group and compare test aggregates and classes, review test results to date, and check test coverage. Blank fields signify incomplete planning and test goals with different "iteration added" or "iteration updated" field values from linked operator/data stream namesake fields signify possible test plan conflicts requiring resolution. Finally, TGTS outputs provide the checklists for ensuring a complete requirements-based test plan. TGTS output samples are included and explained in Appendix A.

D. PERFORMANCE OF THE TEST GOAL TRACKING SYSTEM

Since major portions of CAPS require continued research, TGTS has not been used with CAPS in an actual prototype development. Since this situation will likely persist for the short term, TGTS will not receive a true "in situ" test for some time. TGTS is simple enough to be used for research in parallel with continued research in CAPS prototyping simulations. The TGTS should provide an effective, initial requirements-based prototype testing tool and a tool to help evaluate CAPS-developed prototypes.

TGTS should be viewed as a sort of prototype itself. With continued research, the final form for TGTS' successors will be determined. TGTS will likely be improved by reimplementing in a high level language like C++ or Ada so that it can be directly integrated into the CAPS environment. More detailed functionality and a command line capability to manipulate TGTS would be welcomed additions to make the tool more responsive to the experienced tool user. TGTS experience will establish needed modifications for improved tool performance.

V. CONCLUSIONS AND RECOMMENDATIONS

This research is the initial investigation of software testing to support evolutionary iterative rapid prototyping. As such, it has concentrated on forming and expanding a general methodology (Spiral Testing Method) for conducting software testing and more especially for conducting requirements-based testing in evolutionary iterative rapid prototyping environments. The research further demonstrates the feasibility of testing tool development to support both prototype testing and the Spiral Testing methodology developed herein. There was no attempt to prove rigorously the testing methodology or the TGTS testing tool developed. Such tests will not be possible until prototyping environments and supporting test tools have matured considerably. However, the utility of the tool was demonstrated using published descriptions of a military system.

This thesis addresses several key aspects of software testing to support evolutionary iterative prototyping. It explains the need for a software testing methodology to support the rapid prototyping paradigm and addresses the issue of which prototyping paradigms are most in need of testing support. Additionally, the thesis shows the criticality of requirements-based testing for generating reliable production code from prototyping. Key characteristics of the prototyping-testing relationship are identified. The requirements-based testing tool, TGTS, provides a working research tool for further requirements-based testing research.

TGTS further shows that a series of testing tools, integrated into the prototyping environment can provide increased testing assistance in a non-intrusive way in the prototyping paradigm.

A. RESEARCH CONTRIBUTIONS

This research makes several contributions both to prototyping and to software testing. First, it helps prototyping research efforts. The thesis identifies the need for testing support for evolutionary iterative rapid prototyping. It then identifies particular prototyping characteristics, such as reusable components and prototyping languages, that can aid the testing process. This will allow further prototyping research to consider testing concerns and plan to support them. The intended result will be decreased risk and increased product reliability. In addition, by incorporating the identified characteristics in prototyping environments, researchers can accelerate the development of a production system by prototyping.

Second, this research helps testing. It describes a testing methodology, Spiral Testing, specifically designed to support prototype development in as non-intrusive a way as possible. Spiral Testing is a general testing methodology, tailored expressly for testing in evolutionary iterative rapid prototyping environments. It is general and will support many testing methodologies, taking advantage of the iterative nature of prototyping.

Third, this research helps both prototyping and testing together. The thesis argues the importance of explicit requirements-capturing to both prototyping and requirements-based testing. It then identifies how

the prototyping model and requirements-based testing can be conjoined to harness the requirements-capturing capabilities of prototyping in a way that directly supports requirements-based testing. One of the greatest values of this conjoining is the assurance that testing covers the software buyer's requirements. Another significant value is ensuring prototyping and requirements-based testing processes support each other.

Fourth, this research makes its assertions practical by providing a requirements-based testing tool, TGTS, for use with an existing research prototyping environment, CAPS. The tool assumes a Spiral Testing methodology and evolutionary iterative rapid prototyping. It provides the first tool in an anticipated family of prototyping testing tools.

B. FUTURE RESEARCH

The result of this thesis is to provide a foundation for further prototype testing research. In the prototyping area, more research is needed on assumption recording. Research to make more of the information captured in the prototyping process recoverable for testing purposes would be useful. Prototyping is new enough that many mechanisms needed for truly evolutionary iterative prototyping environments are only in the early stages of development. Tool integration research is essential to incorporate testing tools in prototyping environments. The issue of maintaining test histories on reusable components and component performance within developed prototypes needs attention. Incorporation of other testing methodologies within prototyping, to include structural testing and general functional testing should be studied. Tool support for test condition selection, test oracle derivation, and test execution

and analysis also need research to adapt them to prototyping and Spiral Testing.

At a more specific level, research on how to incorporate consistency checking in testing tools such as TGTS would be invaluable. Further testing automation to include aggregation support also would help prototype testing. With the use of reusable components in prototyping, the ability within CAPS to analyze accompanying component test histories for the purposes of test goal analysis and test case selection will be an important future research area. Shimeall covers additional testing research concerns that pertain particularly to CAPS and PSDL [Shimeall190: p.9].

APPENDIX A TGTS DATABASE SAMPLES

Appendix A takes the prototype example used in Chapter IV and provides sample TGTS outputs to show how TGTS aids requirements-based testing. The example divides outputs into three hypothetical development iterations to show how users employ the tool throughout the prototyping process. Additionally, the example augments the C3I Station requirements with dummy requirements and dummy PSDL operators to illustrate various TGTS capabilities.

A. DATABASE AT CONCLUSION OF FIRST ITERATION

The following figures show TGTS database contents at the conclusion of the first development iteration. In each database's case, the brief output format and the detailed output format are provided for comparative purposes. Note that only one link was established during the first iteration. Testers will need to use discretion to determine when to establish links, so that they point to requirements implementations in ways meaningful to test purposes.

OP_NUMBER	OP_NAME	OPERATOR	UPDATED	O_ITERADD	O_HISTORY	DELETED
1	MP	.T.	0	1	Memo	0
2	MANAGE RADAR TRACKS	.T.	0	1	Memo	0
3	MANAGE DIRECTIVES	.T.	0	1	Memo	0
4	MANAGE PARTICIPATING UNITS	.T.	0	1	Memo	0
5	FORWARD DT	.T.	0	1	Memo	0
6	FORWARD AT	.T.	0	1	Memo	0
7	FORWARD ST	.T.	0	1	Memo	0
8	ASSIGN TRACKS	.T.	0	1	Memo	0

Figure A-1. Iteration One Operator/Data Stream Database Brief Output

The above figure is handy for reviewing operator status and recapping the history of operator and data stream development. The following figure gives the explanation of each operator's or data stream's purpose.

OPERATORS ADDED IN ITERATION 1

op_number op_name
1 MP

o_history
First level module
decomposition of C3I
system.

2 MANAGE RADAR TRACKS

First level module
decomposition of MP,
designed to implement
requirements 1-7 of
initial specification.

3 MANAGE DIRECTIVES

First level decomposition
of MP.

4 MANAGE PARTICIPATING UNITS

First level decomposition
of MP.

5 FORWARD DT

Second level decomposition
of MP.
First level decomposition
of Manage Radar Tracks.

6 FORWARD AT

Second level decomposition
of MP.
First level decomposition
of Manage radar tracks.

7 FORWARD ST

Second level decomposition
of MP.
First level decomposition
of Manage Radar Tracks.

8 ASSIGN TRACKS

Second level decomposition
of MP.
First level decomposition
of Manage Radar Tracks.

Figure A-2. Iteration One Operator/Data Stream Database Descriptive
Output

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
1	Memo	0	0	0	1	4	Memo	0
2	Memo	0	0	0	1	4	Memo	0
3	Memo	0	0	0	1	4	Memo	0
4	Memo	0	0	0	1	4	Memo	0
5	Memo	0	0	0	1	4	Memo	0
6	Memo	0	0	0	1	4	Memo	0
7	Memo	0	0	0	1	4	Memo	0
8	Memo	0	0	0	1	4	Memo	0
9	Memo	0	0	0	1	4	Memo	0
10	Memo	0	0	0	1	4	Memo	0
11	Memo	0	0	0	1	4	Memo	0

Figure A-3. Iteration One Test Goal Database Brief Output

TEST GOALS ADDED IN ITERATION 1		
goal_num	goal_descr	g_history
1	A DT report from DS shall, if possible, be assigned a free or lowest-priority track-id (if the current id is null).	Test goal is decomposed from two cases covered in requirement 1 of initial specifications.
2	Each ST that is broadcast over SL shall be routed to DS if it is consistent with the receiving ship's current tracks; otherwise one of the two inconsistent track reports shall be selected (to become the current track with that id) and routed to DS. The ST is analogous to a remote track in NTDS link-11 capable ships.	Test goal restates requirement 2 of initial specification document.
3	In the absence of communication failures, a concurrent allocation of the same SL track id to two inconsistent tracks by different ships shall be resolved.	Test goal restates requirement 3 of initial system specification. Test goal will need more definition on how inconsistent tracks are resolved.

Figure A-4. Iteration One Test Goal Database Descriptive Output

- | | |
|---|---|
| <p>4 In the absence of communication failures, a concurrent assumption of reporting responsibility for the same SL track shall be resolved.</p> | <p>Test goal restates requirement 4 of initial specification. More details on resolution criteria are needed and will probably require additional test goals.</p> |
| <p>5 A base unit MP shall route to AL those SL tracks with ids that are designated by DS (with TA messages) for reporting over AL (irrespective of which ship has reporting responsibility). This routing shall persist until a TA message retracts it or the track is dropped.</p> | <p>Test goal restates requirement 5 in original specification document.</p> |
| <p>6 Each track report in a packet received from a ship shall be processed or discarded within one link cycle of its arrival at the MP. That is, before another updated or new track report can arrive from the same unit.</p> | <p>Test goal is decomposed from requirement 6 in the initial specification. Ships and aircraft are distinct. User comment made this clear.</p> |
| <p>7 Each track report received from an aircraft shall be processed or discarded within one link cycle of its arrival at the MP. That is, before another updated or new track can arrive from the same unit.</p> | <p>Test goal is decomposed from requirement 6 in the initial specification. Ships and aircraft are distinct. This was made clear by user comment.</p> |
| <p>8 Each DT message from DS to MP shall be transmitted over SL within two link cycles of its arrival at MP if there are 255 or less words of message data to be transmitted in the next outgoing packet.</p> | <p>Test goal is decomposed from two cases covered in requirement 7 of initial specification.</p> |

Figure A-4(cont'd). Iteration One Test Goal Database Descriptive Output

- | | |
|--|--|
| <p>9 Each DT message from DS to MP need not be transmitted over SL within two link cycles of its arrival at MP if there are more than 255 words of message data in the next outgoing packet. In that event, queued messages shall be deleted or replaced in order of priority until the next transmission opportunity. The message handling priorities, in descending priority are:</p> <ul style="list-style-type: none"> - ISL, IAL, and ALS. - incoming and outgoing directives. - outgoing messages (from base units) to remote units (aircraft). - incoming SL and AL track reports. - outgoing SL track reports. - all other messages (including responses to directives). | <p>Test goal is decomposed from two cases in requirement 7 of the initial specification.</p> |
| <p>10 A DT report from DS shall be broadcast over SL within two SL cycles of its arrival at the MP.</p> | <p>Test goal is decomposed from two cases covered in requirement 1 of the initial specification.</p> |
| <p>11 Minimum ship link cycle duration shall be 500 ms.</p> | <p>Derived test goal. Source is initial specification requirement 22 relating to the DS module.</p> |

Figure A-4 (cont'd). Iteration One Test Goal Database Descriptive Output

GOAL_NUM	OP_NUMBER	PSDL_PART	L_ITERADD	DELETED
11	5	OPERATOR - ALL	1	0

Figure A-5. Iteration One Link Database Output

B. DATABASE AT CONCLUSION OF SECOND ITERATION

The following database printouts reflect the additions and changes resulting from the second iteration's development. Links and requirements changes are incorporated into TGTS' database. The figure below of the test goal database reflects the removal of test goal two and its replacement by test goal fourteen. Additionally, Test Goals Twelve and Thirteen were added in the second iteration.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
1	Memo	0	0	0	1	4	Memo	0
3	Memo	0	0	0	1	4	Memo	0
4	Memo	0	0	0	1	4	Memo	0
5	Memo	0	0	0	1	4	Memo	0
6	Memo	0	0	0	1	4	Memo	0
7	Memo	0	0	0	1	4	Memo	0
8	Memo	0	0	0	1	4	Memo	0
9	Memo	1	1	2	1	4	Memo	0
10	Memo	0	0	0	1	4	Memo	0
11	Memo	0	0	0	1	4	Memo	0
12	Memo	0	0	0	2	2	Memo	0
13	Memo	0	0	0	2	2	Memo	0
14	Memo	0	0	0	2	4	Memo	0

Figure A-6. Iteration Two Test Goal Database Brief Output

Figure A-7 gives the new portion of the test goal database. Note that for Test Goal Fourteen, the goal history field gives Test Goal Two as its precursor. Test Goals Twelve and Thirteen provide examples of new test goals resulting from user input during a prototype demonstration.

TEST GOALS ADDED IN ITERATION 2

goal_num	goal_descr	g_history
12	Test goal 12. A dummy for example purposes.	Test goal restates requirement 25, recorded from iteration 1 demonstration user comment.
13	Test goal 13. Dummy test goal for example.	Test goal restates requirement 26, recorded from iteration 1 demonstration user comment.
14	Each ST that is broadcast over SL shall be routed to DS if it is consistent with the receiving ship's current SL tracks; otherwise a correlation conflict warning shall be sent to the DS. Both tracks shall be selected and routed to the DS for resolution by the system operator. The system operator must have control over traffic conflicts. MP conflict resolution recommendations may be forwarded to the DS to assist the operator.	Replaces test goal 2. Restates initial specification requirement 2. User provided amplifying information explaining inconsistent track resolution.

Figure A-7. Iteration Two Added Test Goals Descriptive Output

An example listing of deleted goals is contained in the following figure . Observe that its history field reflects its successor test goal, allowing a test goal history trace.

TEST GOALS DELETED IN ITERATION 2

goal_num	goal_descr	g_history
2	Each ST that is broadcast over SL shall be routed to DS if it is consistent with the receiving ship's current tracks; otherwise one of the two inconsistent track reports shall be selected (to become the current track with that id) and routed to DS. The ST is analogous to a remote track in NTDS link-11 capable ships.	Test goal restates requirement 2 of initial specification document. -Deleted in iteration 2. Replaced by test goal 14.

Figure A-8. Iteration Two Deleted Test Goals Output

The added PSDL modules are reflected below. Operators "V," "X," "Y," and "Z" decompose the Forward DT module. Operators "A," "B," "C," and "D" decompose the Assign Track module.

OP_NUMBER	OP_NAME	OPERATOR	UPDATED	O_ITERADD	O_HISTORY	DELETED
9	W	.T.	0	2	Memo	0
10	X	.T.	0	2	Memo	0
11	Y	.T.	0	2	Memo	0
12	Z	.T.	0	2	Memo	0
13	A	.T.	0	2	Memo	0
14	B	.T.	0	2	Memo	0
15	C	.T.	0	2	Memo	0
16	D	.T.	0	2	Memo	0

Figure A-9. Iteration Two Operators Added Brief Output

OPERATORS ADDED IN ITERATION 2
 op_number op_name
 9 W

o_history
 Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Forward DT.

10 X

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Forward DT.

11 Y

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Forward DT.

12 Z

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Forward DT.

13 A

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Assign Track.

14 B

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Assign Track.

15 C

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Assign Track.

16 D

Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Assign Track.

Figure A-10. Iteration Two Operators Added Descriptive Output

In the second iteration enough functionality had been added to the prototype to allow the test team to begin linking test goals to their implementation. The following figure shows the links established.

GOAL_NUM	OP_NUMBER	PSDL_PART	L_ITERADD	DELETED
1	13	AXIOM	2	0
1	14	AXIOM	2	0
1	15	AXIOM	2	0
1	16	AXIOM	2	0
9	11	AXIOM	2	0
9	12	AXIOM	2	0
9	13	IMPLEMENTATION ADA	2	0
9	14	AXIOM	2	0
11	9	AXIOM	2	0
11	10	AXIOM	2	0
12	15	AXIOM	2	0
13	16	AXIOM	2	0

Figure A-11. Iteration Two Link Database Output

The next two figures show how the links are useful to cross reference. First, Figure A-12 shows how a particular goal number can be flagged to trace all PSDL parts that implement the testable behavior. In this case, Test Goal Eleven is flagged to see which PSDL operators affect it. Operators Nine and Ten implement Test Goal Eleven, according to the links and they are a decomposition of Operator 5. Then Figure A-13 shows how an operator can be flagged to see which test goals are related to it.

GOAL_NUM	OP_NUMBER	PSDL_PART	L_ITERADD	DELETED
11	5	OPERATOR - ALL	1	0
11	9	AXIOM	2	0
11	10	AXIOM	2	0

Figure A-12. Links to a Particular Test Goal Sample Output

GOAL_NUM	OP_NUMBER	PSDL_PART	L_ITERADD	DELETED
1	15	AXIOM	2	0
12	15	AXIOM	2	0

Figure A-13. Links to a Particular Operator Sample Output

Modified test goals can easily be examined. In our hypothetical example, the test team has modified Test Goal Nine's record to reflect that it is prioritized for testing and has a test oracle data set under construction. Figures A-14 and A-15 shows the TGTS outputs for this test goal.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
9	Memo	1	1	2	1	4	Memo	0

Figure A-14. TGTS Brief Output for a Modified Test Goal

TEST GOALS MODIFIED IN ITERATION 2

goal_num	goal_descr	g_history
9	Each DT message from DS to MP need not be transmitted over SL within two link cycles of its arrival at MP if there are more than 255 words of message data in the next outgoing packet. In that event, queued messages shall be deleted or replaced in order of priority until the next transmission opportunity. The message handling priorities, in descending priority are:	Test goal is decomposed from two cases in requirement 7 of the initial specification.
	<ul style="list-style-type: none"> - ISL, IAL, and ALS. - incoming and outgoing directives. - outgoing messages (from base units) to remote units (aircraft). - incoming SL and AL track reports. - outgoing SL track reports. - all other messages (including responses to directives). 	Iteration 2 - prioritized, aggregate started, Test oracle uses test data set 1.

Figure A-15. TGTS Descriptive Output for a Modified Test Goal

C. USE OF TGTS' OUTPUTS FOR TEST PLANNING PURPOSES

TGTS provides numerous database outputs useful to test planners. A sample of these is included to show how they are used for test planning. The example of Chapter IV continues with Iteration Three as a vehicle to examine TGTS' outputs.

In Iteration Three, the test team adds three additional hypothetical test goals into TGTS as decompositions of a Requirement 50 resulting from Iteration Two's user demonstration. These are reflected below as Test Goals Fifteen, Sixteen and Seventeen. The designers modify Requirement Seven to conform to user input as well which results in Test Goal Nine

being superseded by Test Goals Eighteen and Nineteen below. Note that Test Goal Nine is no longer a part of the database output. Also Test Goal Eighteen has been compared with Test Goal Nine and the priority, aggregate and data set remained the same.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
1	Memo	0	0	0	1	4	Memo	0
3	Memo	0	0	0	1	4	Memo	0
4	Memo	0	0	0	1	4	Memo	0
5	Memo	0	0	0	1	4	Memo	0
6	Memo	0	0	0	1	4	Memo	0
7	Memo	0	0	0	1	4	Memo	0
8	Memo	0	0	0	1	4	Memo	0
10	Memo	0	0	0	1	4	Memo	0
11	Memo	0	0	0	1	4	Memo	0
12	Memo	0	0	0	2	2	Memo	0
13	Memo	0	0	0	2	2	Memo	0
14	Memo	0	0	0	2	4	Memo	0
15	Memo	0	0	0	3	4	Memo	0
16	Memo	0	0	0	3	4	Memo	0
17	Memo	0	0	0	3	4	Memo	0
18	Memo	1	1	0	3	4	Memo	0
19	Memo	0	0	0	3	2	Memo	0

Figure A-16. Iteration Three Test Goal Database Brief Output

The test goals added in Iteration Three are shown next. Test Goal Eighteen's goal history field reflects that it replaced Test Goal Nine and is part of a requirement decomposition.

TEST GOALS ADDED IN ITERATION 3

goal_num	goal_descr	g_history
15	Test goal 15. Dummy test goal for example.	Test goal decomposed from requirement 50, recorded from iteration 2 demonstration user comment.
16	Test goal 16. Dummy test goal for example.	Test goal is decomposed from requirement 50, recorded from iteration 2 demonstration user comment.
17	Test goal 17. Dummy test goal for example.	Test goal restates requirement 55, recorded from iteration 2 demonstration user comment.
18	Each DT message from DS to MP need not be transmitted over SL within two link cycles of its arrival at MP if there are more than 255 words of message data in the next outgoing packet. In that event, queued messages shall be deleted or replaced in order of priority until the next transmission opportunity. The message handling priorities, in descending priority are: - ISL, IAL, SLS, and ALS. - Hostile Quick Response Tracks (such as pop-up missiles). - Incoming and outgoing directives, including responses to directives. - Outgoing messages (from base units) to remote units (aircraft). - Incoming and outgoing hostile and unknown DS, SL, and AL track reports. - Incoming and outgoing friendly SL and AL track reports. - All other messages.	Replaces test goal 9. Decomposes initial specification requirement 7, as modified by iteration 2 demonstration user comments. Requirement 7 decomposes into 2 test goals.
19	Hostile Quick Response Tracks shall be identified by a special field in the track descriptor.	Decomposed from modified requirement 7. User comments in iteration 2 demonstration caused modifications. Two test goals derived, 18 and 19.

Figure A-17. Iteration Three Test Goals Added

In furthering the prototype's functionality, developers have added modules "U" and "V" to provide the source code for Operator "Y". In addition, designers modified modules (operators) "Y," "Z," "A," and "B" to account for changes in Requirement Seven. These developments are shown in Figure A-18 and Figure A-19 as an excerpt of the Operator/Data Stream Database output for the third iteration.

OP_NUMBER	OP_NAME	OPERATOR	UPDATED	O_ITERADD	O_HISTORY	DELETED
1	MP	.T.	0	1	Memo	0
2	MANAGE RADAR TRACKS	.T.	0	1	Memo	0
3	MANAGE DIRECTIVES	.T.	0	1	Memo	0
4	MANAGE PARTICIPATING UNITS	.T.	0	1	Memo	0
5	FORWARD DT	.T.	0	1	Memo	0
6	FORWARD AT	.T.	0	1	Memo	0
7	FORWARD ST	.T.	0	1	Memo	0
8	ASSIGN TRACKS	.T.	0	1	Memo	0
9	W	.T.	0	2	Memo	0
10	X	.T.	0	2	Memo	0
11	Y	.T.	3	2	Memo	0
12	Z	.T.	3	2	Memo	0
13	A	.T.	3	2	Memo	0
14	B	.T.	3	2	Memo	0
15	C	.T.	0	2	Memo	0
16	D	.T.	0	2	Memo	0
17	U	.T.	0	3	Memo	0
18	V	.T.	0	3	Memo	0

Figure A-18. Iteration Three Operator/Data Stream Database Extract

OPERATORS CHANGED IN ITERATION 3
 op_number op_name
 11 Y

o_history
 Second level decomposition
 of Manage Radar Tracks.
 First level decomposition
 of Forward DT.

Iteration 3 - modified to
 account for requirement 7
 modification from
 iteration 2 demonstration
 user comment.

Figure A-19. Iteration Three Operator Database Detailed Extract

12 Z

Second level decomposition
of Manage Radar Tracks.
First level decomposition
of Forward DT.

Iteration 3 - modified to
account for requirement 7
changes from iteration 2
demonstration user
comments.

13 A

Second level decomposition
of Manage Radar Tracks.
First level decomposition
of Assign Track.

Iteration 3 - checked, not
modified from initial
requirement 7 after
iteration 2 demonstration
user comments caused
requirement 7
modification.

14 B

Second level decomposition
of Manage Radar Tracks.
First level decomposition
of Assign Track.

Iteration 3 - updated to
reflect requirement 7
modifications from
iteration 2 demonstration
user comments.

Figure A-19 (cont'd). Iteration Three Operator Database Detailed Extract

The modified operators were relinked to the modified test goal (Eighteen) and the links to the old test goal (Nine) were deleted. The changes in the operators mentioned in the above paragraphs resulted in the test team determining further links. The following figure shows the link database at some intermediate point in Iteration Three.

GOAL_NUM	OP_NUMBER	PSDL_PART	L_ITERADD	DELETED
1	13	AXIOM	2	0
1	14	AXIOM	2	0
1	15	AXIOM	2	0
1	16	AXIOM	2	0
11	5	OPERATOR - ALL	1	0
11	9	AXIOM	2	0
11	10	AXIOM	2	0
12	15	AXIOM	2	0
13	16	AXIOM	2	0
15	17	AXIOM	3	0
15	18	AXIOM	3	0
16	17	AXIOM	3	0
16	18	AXIOM	3	0
17	17	AXIOM	3	0
17	18	AXIOM	3	0
18	11	AXIOM	3	0
18	12	AXIOM	3	0
18	13	IMPLEMENTATION ADA	3	0
18	14	AXIOM	3	0

Figure A-20. Iteration Three Link Database Output

The following figure shows the deleted links for Iteration Three.

GOAL_NUM	OP_NUMBER	PSDL_PART	L_ITERADD	DELETED
9	11	AXIOM	2	3
9	12	AXIOM	2	3
9	13	IMPLEMENTATION ADA	2	3
9	14	AXIOM	2	3

Figure A-21. Iteration Three Deleted Links

As test plan development continues through an iteration, the test team modifies the TGTS databases first to reflect changes in requirements that affect database test goals. Then they add prototype coding changes, comparing changes to modules that TGTS indicates may have been affected by requirements changes. The test team also modifies links accordingly. They check any new modules for needed linking to existing or new test goals. Testers match any new test goals against existing modules for linking. They check all links of any deleted test goals to see if they point to operators/data streams that need to be linked to new or other

modules. Finally, testers check deleted operators/data streams to see which new operators/data streams replace them and then relink them accordingly.

The end result of checking, adding and deleting links is an examination of each part of the test plan that links affect. This is followed by necessary changes to keep the test plan current with the existing test goals and the prototype implementation. The changes manifest themselves as updates to the test goal records, reflecting test priority assignment, aggregating, and test history comments to link test goals to particular test oracle data sets and test results as appropriate. The following figures reflect how the example TGTS test goal database might look after a bit of such work by testers in Iteration Three. They include several of the output groupings for the test goals database.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_H'STORY	DELETED
1	Memo	0	0	0	1	4 Memo		0
3	Memo	0	0	0	1	4 Memo		0
4	Memo	0	0	0	1	4 Memo		0
5	Memo	0	0	0	1	4 Memo		0
6	Memo	0	0	0	1	4 Memo		0
7	Memo	0	0	0	1	4 Memo		0
8	Memo	0	0	0	1	4 Memo		0
10	Memo	0	0	0	1	4 Memo		0
11	Memo	1	1	3	1	4 Memo		0
12	Memo	0	0	0	2	2 Memo		0
13	Memo	0	0	0	2	2 Memo		0
14	Memo	0	0	0	2	4 Memo		0
15	Memo	2	2	3	3	4 Memo		0
16	Memo	2	2	3	3	4 Memo		0
17	Memo	1	2	3	3	4 Memo		0
18	Memo	1	1	0	3	4 Memo		0
19	Memo	1	0	3	3	2 Memo		0

Figure A-22. Iteration Three Test Goal Modifications

Listings for particular test classes can be produced as shown next.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
12	Memo	0	0	0	2	2 Memo		0
13	Memo	0	0	0	2	2 Memo		0
19	Memo	1	0	3	3	2 Memo		0

Figure A-23. Test Class 2 Test Goals

Test goals may also be listed by priority. This output is helpful for planning the order in which to conduct the testing.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
9	Memo	1	1	2	1	4 Memo		3
18	Memo	1	1	0	3	4 Memo		0
11	Memo	1	1	3	1	4 Memo		0
17	Memo	1	2	3	3	4 Memo		0
19	Memo	1	0	3	3	2 Memo		0

Figure A-24. Test Priority 1 Test Goals

The following output allows testers to view all the test goals for a given aggregate. A detailed output is available for all the above test goal outputs as well as the output below.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
9	Memo	1	1	2	1	4 Memo		3
18	Memo	1	1	0	3	4 Memo		0
11	Memo	1	1	3	1	4 Memo		0

Figure A-25. Test Aggregate 1 Test Goals

Testers will often need to review and recap the updates made during a given iteration. The following figure is a sample output of the modified test goals for the third iteration.

GOAL_NUM	GOAL_DESCR	TEST_PRI	AGGREGATE	UPDATED	ITER_ADDED	TEST_CLASS	G_HISTORY	DELETED
11	Memo	1	1	3	1	4 Memo		0
15	Memo	2	2	3	3	4 Memo		0
16	Memo	2	2	3	3	4 Memo		0
17	Memo	1	2	3	3	4 Memo		0
19	Memo	1	0	3	3	2 Memo		0

Figure A-26. Iteration Three Modified Test Goals Brief Output

The final output example is the descriptive output of the modified test goals for Iteration Three. The example, in a real development and testing process would continue until all test goals were developed and all tests conducted. The database would continue to be loaded to reflect the test plan, and its results through to the completion of all testing.

TEST GOALS MODIFIED IN ITERATION 3

goal_num	goal_descr	g_history
11	Minimum ship link cycle duration shall be 500 ms.	Derived test goal. Source is initial specification requirement 22 relating to the DS module. Iteration 3 - Update. Test oracle uses test data set 2.
15	Test goal 15. Dummy test goal for example.	Test goal decomposed from requirement 50, recorded from iteration 2 demonstration user comment.
16	Test goal 16. Dummy test goal for example.	Iteration 3 - Updated. Test Oracle uses test data set 5. Test goal is decomposed from requirement 50, recorded from iteration 2 demonstration user comment.
17	Test goal 17. Dummy test goal for example.	Iteration 3 - Updated. Test oracle uses test data set 6. Test goal restates requirement 55, recorded from iteration 2 demonstration user comment.
		Iteration 3 - Updated. Test oracle uses test data sets 3 & 4.

Figure A-27. Iteration Three Modified Test Goals Descriptive Output

<p>19 Hostile Quick Response Tracks shall be identified by a special field in the track descriptor.</p>	<p>Decomposed from modified requirement 7. User comments in iteration 2 demonstration caused modifications. Two test goals derived, 18 and 19.</p> <p>Iteration 3 - Updated. Test priority assigned.</p>
---	--

Figure A-27. Iteration Three Modified Test Goals Descriptive Output

The Iteration History outputs are not shown since they have no prescribed format and consist simply of the iteration number and a text field for recording the summary of an iteration's test-related history.

APPENDIX B TGTS USER'S MANUAL

The TGTS User's Manual provides user guidance for the Test Goal Tracking System (TGTS). TGTS is a requirements-based testing tool for use with the Computer Aided Prototyping System (CAPS) at the Naval Postgraduate School. The tool is a database-type tool that allows testers to record explicitly both test goals and related test plan information concerning prototypes developed with CAPS. Test goals can be explicitly linked to the portions of the prototype's PSDL code or the implementation language modules that produce the final production code. Additionally, test information for each iteration may be summarized in an iteration information database for reference purposes.

Tgts is a menu-driven tool with nested menus for the user. Menu selection is by numerical selection of a menu item, thus TGTS is an easy tool to begin to use.

This manual assumes the reader is familiar with three things:

- CAPS and PSDL and the associated prototyping process
- Spiral Testing, as described by Davis in his 1990 NPS Thesis
- dBase III+

Additionally, TGTS is a research tool for testing in evolutionary iterative rapid prototyping. As such, it is to be used, modified and eventually reimplemented in a high level language. The reimplementation of TGTS will allow it to become the first of a family of integrated

testing tools within the CAPS prototyping environment. CAPS is currently under development at the Naval Postgraduate School.

A. TGTS STRUCTURE AND DATABASE FIELDS

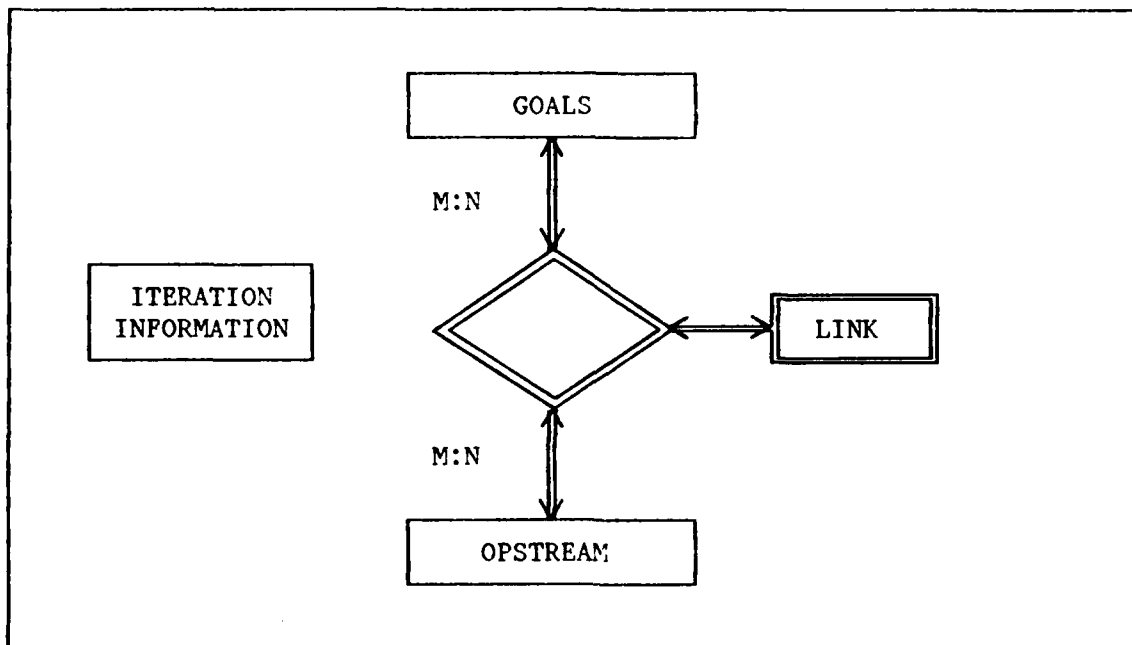
1. TGTS Structure

TGTS is a database tool implemented in dBase III+. It is not compiled, so it must be run from within dBase III+, which is a DOS software application. TGTS has three main databases:

- Test Goal Database - for storing test goals and related testing information
- Operator/Data Stream Database - for storing information on all PSDL parts that implement the prototype undergoing development and testing
- Link Database - for storing the records that link particular test goals to particular operators/data streams

There is also another database, the Iteration Information Database, that does not link with the other three databases. It is a summary or "catch-all" database for recording such iteration related information as testers deem necessary.

The mappings between the "Goals" and "Opstream" database may be one-to-many, many-to-one and many-to-many. Usually they will be many-to-many. The following figure illustrates the database structure for TGTS.



The database fields are excluded from the above figure to avoid clutter. The following table lists the fields and field characteristics of the database records and includes the dBase III+ field types and index files relating to fields. DBase keeps the index files listed to index records. Each index file's key is the respective field name beside which the index file is listed.

TABLE B-1

GOALS.DBF

Field	Field Name	Type	Width	Range	Index File Name
1*	Goal Number	Numeric	6	1-9999	GOAL_NUM.NDX
2	Goal Description	Memo			
3	Test Priority	Numeric	1	0-4	TEST_PRI.NDX
4	Aggregate	Numeric	2	0-99	AGGREGAT.NDX
5	Updated	Numeric	2	0-99	UPDATED.NDX
6	Iteration Added	Numeric	2	1-99	ITER_ADD.NDX
7	Test Class	Numeric	1	0-4	TESTCLAS.NDX
8	Goal History	Memo			
9	Deleted	Numeric	2	0-99	DELET_TG.NDX

TABLE B-1 (CONT'D)

OPSTREAM.DBF

Field	Field Name	Type	Width	Range	Index File Name
1*	Operator Number	Numeric	6	1-9999	OP_NUM.NDX
2	Operator Name	Character	40		OP_NAME.NDX
3	Operator	Logical	1	T or F	
4	Updated	Numeric	2	1-99	O_UPDATED.NDX
5	O Iteration Added	Numeric	2	1-99	O_ITERAD.NDX
6	Operator History	Memo			
7	Deleted	Numeric	2	0-99	DELET_OP.NDX

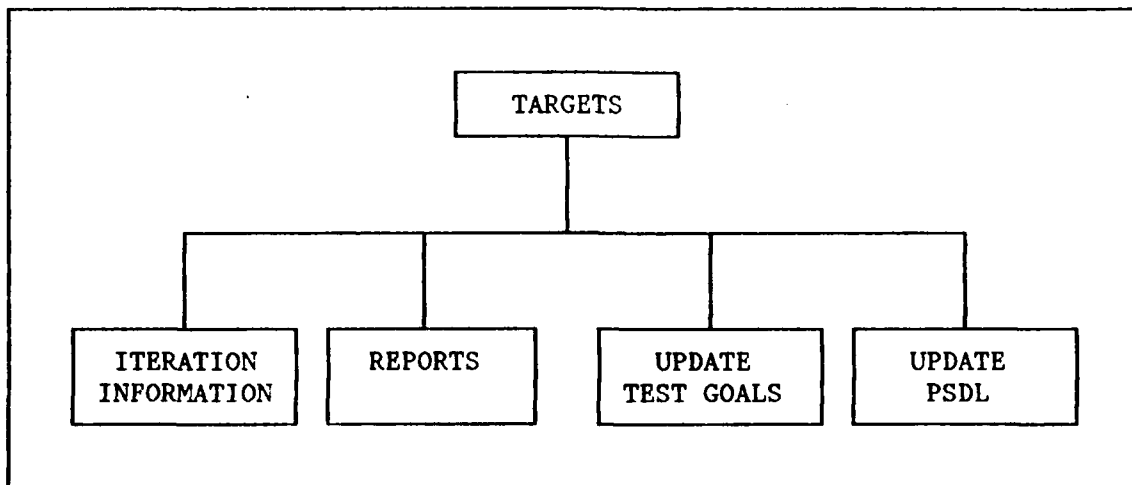
RO LINK.DBF

Field	Field Name	Type	Width	Range	Index File Name
1*	Goal Number	Numeric	6	1-9999	GNUMLINK.NDX
2*	Operator Number	Numeric	6	1-9999	ONUMLINK.NDX
3	PSDL Part	Character	40		PARTLINK.NDX
4	L Iteration Added	Numeric	2	1-99	L_ITERAD.NDX
5	Deleted	Numeric	2	0-99	DELET_LK.NDX

* - INDICATES KEY FIELD

Note that each of the first two databases assigns a unique number to each record that serves as a key for referencing the record. This allows the link database to establish unique links simply by referencing the two numbers. The field names in Table B-1 are written out descriptively. The field names in the source code are shorter to fit dBase programming requirements. The ranges chosen for the fields are arbitrary and provide sufficient range for anticipated projects.

TGTS has a modular design. The following figure shows the top level program decomposition. The decomposition groups logically related tool functions together.



For example, the "Reports" module contains all the output functions for TGTS and the "Update Test Goals" module contains the functions that manipulate the test goals database.

2. TGTS Database Fields

This subsection describes the database fields for each TGTS database, including field use and limitations. For the dBase features, refer to Table B-1.

a. Test Goal Database Fields

The test goals database (GOALS.DBF in the implementation) has nine fields.

- **Goal Number** - assigned automatically and sequentially, in the order of record entry, by the tool. The value is inaccessible for change. It serves as the key field for the database for test goal record retrieval. It allows a maximum of 9999 test goals.
- **Goal Description** - the English text stating the test goal. This is the field for storing each test goals. The size of the text allowed is unlimited if it is to be manipulated by a word processor, but dBase can only handle 5000 characters with the MODIFY COMMAND editor.

- Test Priority - for recording the priority assigned to testing the test goal. Legal values are 0-4 with '0' signifying unassigned. Priority '1' is the highest priority for testing.
- Aggregate - for storing the test aggregate encompassing the test goal. Legal values are 0-99 with '0' signifying unassigned. Assumes a maximum of 99 aggregates in the total test plan.
- Updated - for storing the last iteration number in which the user updated the test goal to reflect a change in any of fields 3-8. Any previous updates will be overwritten in this field. Legal values are 0-99 with '0' signifying unassigned.
- Iteration Added - for recording the prototype iteration in which the test goal was added. Allows testers to track the history of when test goals were added so that they can be compared to changes in the prototype's implementation. This field should not be left blank.
- Test Class - for recording the test class of a test goal. Legal values are 0-4 with '0' signifying unassigned. The four test classes are 1 - untestable, 2 - observation, 3 - analysis, and 4 - execution.
- Goal History - for recording text descriptions of the test goals history to include: source of derivation (requirements numbers, document, user comments - be specific), updates undertaken, to include iteration and key points of update, reference to test data sets and test oracle references to link the test goal to particular test plan portions, and test results and other notes deemed necessary for testers. The text limitations are identical with those for goal description. For deleted goals only, prior to deletion, enter a change to the history to reflect the reasons for the deletion and list any new or other test goals that will replace or supercede the deleted test goal. For undeleted records, the reason for undeletion should be recorded.
- Deleted - for automatically recording the iteration number in which testers deleted the test goal. Legal values are 0-99 with '0' signifying the test goal is not deleted. Prior to deletion of test goals, testers should update the goal history to reflect the deletion's purpose. Should a test goal be undeleted, the user should insert the iteration of undeletion and updates the record's history to reflect undeletion.

Test goal descriptions are never changed. If the description is invalid then the entire test goal should be deleted. Invalid test goals are those superseded by multiple, more specific test goals, those that have become obsolete due to prototype changes or those that are incorrect.

b. Opstream Database Fields

The Opstream Database (OPSTREAM.DBF in the implementation) has seven fields.

- Operator Number - assigned automatically and sequentially, in the order of record entry, by the tool. The value is inaccessible for change. It serves as the key field for the database for opstream record retrieval. It allows a maximum of 9999 records.
- Operator Name - for recording the name of the PSDL operator or data stream that implements the test goal. All legal dBase "character" characters, up to 40 in length are allowed. The tool converts all into upper case.
- Operator - for recording whether the PSDL part is an operator or a data stream. This field is a boolean flag and accepts 'T' or 'F' or 'Y' or 'N'.
- Updated - for storing the last iteration number in which the user updated the PSDL part to reflect a change in any of fields 2-4 or 6. Any previous updates will be overwritten in this field. Legal values are 0-99 with '0' signifying unassigned.
- 0 Iteration Added - for recording the prototype iteration in which developers added the PSDL part (opstream). It allows testers to track the history of when developers added operators and data streams so that the implementation can be compared to test goals that test it. This field should not be left blank.
- Operator History - for recording text descriptions of the opstream's history to include: source of derivation (requirements numbers, document, user comments - be specific), updates undertaken, to include iteration and key points of update, reference to iterations and test results to link the module to particular test plan portions, and test results and other notes deemed necessary for testers. The text limitations are 5000 characters for use with the MODIFY COMMAND editor but no size restriction exists with word processors. For deleted opstream records only, prior to deletion, enter a change to the history to reflect the reasons for the deletion and list any new or other opstreams that will replace or supercede the deleted one. For undeleted records, the reason for undeletion should be recorded.
- Deleted - for automatically recording the iteration number in which developers deleted the opstream record. Legal values are 0-99 with '0' signifying the opstream is not deleted. Prior to deletion of opstreams, the tester updates the operator history to reflect the deletion's purpose. Should an opstream be undeleted, the user should

insert the iteration of undeletion and update the record's history to reflect undeletion.

c. Link Database Fields

The Link Database (RO_LINK.DBF in the implementation) has five fields. Two of these fields are key fields and serve to create a link between records of the Test Goals and Opstream Databases.

- Goal Number - as in goals database, but the user manually enters data to match the desired link. Legal values are 1-9999. This field must not be left blank. Responsibility for correct link numbers is the users. It is one of two key fields.
- Operator Number - as in opstream database, but data is manually entered to match the desired link. Legal values are 1-9999. This field must not be left blank. Responsibility for correct link numbers is the users. This is the second key field.
- PSDL Part - for entering the PSDL correct grammatical part name when the test goal relates to an operator's or data stream's internal part. Any legal dBase "character" character is legal. Responsibility for correct name match is the user's.
- L Iteration Added - for recording the prototype iteration in which the user added the link record. It allows testers to track the history of when operators and data streams were linked so that they can be compared to test goals and opstream records. This field should not be left blank.
- Deleted - for automatically recording the iteration number in which the user deleted the link. Legal values are 0-99 with '0' signifying the opstream is not deleted. When a user deletes either a test goal or an opstream record, then TGTS deletes any links to it automatically and inserts the deletion iteration number in the deleted field. Should a user reinstate a link, then TGTS resets the field to '0' automatically.

d. Iteration Information Database Fields

The Iteration Information Database (ITERATNS.DBF in the implementation) records contain only two fields.

- Iteration Number - automatically inserted iteration number for the iteration the user is recording. Legal range is 1-99.
- Iteration History - for recording text concerning the iteration's history as it relates to testing. The user records significant events, (as defined by the user) as text. The MODIFY COMMAND editor can only edit up to 5000 characters but word processors can edit any field length.

TGTS non-destructively deletes all database records so that they can be readded into the database should a prototype iteration fall back to a previous position or a user reinstate a previously removed requirement. The non-destructive delete keeps the database flexible enough to respond to multiple changes.

B. TGTS TOOL OPERATIONS

The primary purpose of TGTS is to record test plan information on a developing prototype's test goals and then map the test goals to their source in the prototype code. As TGTS' top-level module decomposition shows, the operations fall into four categories: database outputs, test goal updates, PSDL updates, and iteration information. The principal operations of each will be discussed in turn. For many operations, sample TGTS' menu screens show the user choices available.

1. Starting TGTS

Users start TGTS by a three step process. First, the user starts the dBase program with the command "DBASE." Second, the user then removes the "ASSIST" menu by pressing <ESC>. Third, the user types in the command "DO TESTGOAL" at the dBase command dot prompt. TGTS commences with a welcome screen, followed by the top-level menu, both shown below.

The brief description excludes the descriptive text fields for each record. The descriptive output includes the text fields and excludes all numeric fields except the key fields.

At the most basic output level are the "All" output options shown below. A user may list all contents for either of the three databases, which is useful in many general ways. In each of these listings, the lists may be indexed by any of several keys to provide users with appropriate record groupings.

[illegible]

More specific listing operations exist. Test goals added, changed or deleted in a given iteration as well as those of a given test class or priority or aggregate may be listed. The "Deleted Test Goals" listing options screen is shown next.


```

/MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:
:      TEST GOALS THAT MAP TO A GIVEN OP/DATA STREAM      :
:
:      Select your choice by number:                      :
:
:      1. List Test Goals for all iterations              :
:      2. List Test Goals for a given iteration           :
:      3. Return to Report Menu                          :
:
:      Enter your choice here:  2                        :
:MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM:
:      Enter the iteration to output or 0 to escape       :
:      Enter your choice here:  2                        :
:MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM:
:      1. Brief output:  description & history omitted    :
:      2. Descriptive output:  w/ description & history   :
:
:      Enter your choice here:  1                        :
:      Enter the op/data stream no. for goal listing:    0 :
/MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```

In similar fashion, operators or data streams added, changed, or deleted in a given iteration may be listed. The two following menu screen examples show how menu screens appear for these type screens.

```

/MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:
:           OPERATOR LISTING OPTIONS           :
:
:   Select your choice by number:              :
:
:   1. List by Operator Number                 :
:   2. List alphabetically                    :
:   3. List deleted operators                  :
:   4. Return to Report Menu                  :
:
:
:
:   Enter your choice here:  1                 :
:MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:   1. Brief output:  description & history omitted :
:   2. Descriptive output:  w/ description & history :
:   Enter your choice here:  1                 :
:
:
/MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```

```

/MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:
:   OPERATORS/DATA STREAMS ADDED FOR A GIVEN ITERATION :
:
:   Select your choice by number:              :
:
:   1. List Operators                         :
:   2. List Data Streams                     :
:
:
:   3. Return to Report Menu                 :
:MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:   Enter your choice here:  1                 :
:
:   Enter the iteration to output:  2         :
:MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:   1. Brief output:  description & history omitted :
:   2. Descriptive output:  w/ description & history :
:   Enter your choice here:  1                 :
:
:
/MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```


The link outputs help provide a history of test development concurrent with the system development process and serves as a test plan management aid. When testers combine link listings with detailed listings of the other databases, they have an overview of the entire requirements-based test plan.

3. TGTS Test Goal Update Operations

[illegible]

a. Add a New Test Goal

This operation adds a new test goal record to the test goals database. Initially, users fill the test goal record's description field. They then fill in the goal history field with a justification of the goal's derivation, such as "user comment on iteration x demonstration" or "initial requirements document, requirement no. 23." The history field entry should be a sufficiently clear description so that the test team knows explicitly where the goal originated.

The "Add" operation automatically assigns test goal numbers sequentially. This prevents two test goals from having identical test goal numbers and allows the numerical value of test goals to reflect the sequence of test goal derivation, should the test team deem such information valuable. The test goal data input screen is shown below.

```

|MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:                                     TEST GOALS DATABASE DATA ENTRY                                     :
|MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<
TEST GOAL NUMBER                      20

```

GOAL DESCRIPTION memo Move cursor to memo field, type Ctrl-PgDn

TEST PRIORITY 0 AGGREGATE 0 TEST CLASS 0

ITERATION ADDED 0

HISTORY memo Move cursor to memo field, type Ctrl-PgDn

```

|MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
: to make corrections,                                                         :
: Return to continue, C to Cancel,                                           :
: or X to exit                                                                :
|MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```

b. Change an Existing Test Goal

The second test goal operation is the "Change" operation. Unlike what one probably thinks, this operation is not intended to allow testers to change the nature and description of a given test goal. The "Change" operation allows testers to change test plan related information concerning a given test goal, such as its test priority or test aggregate. It also allows testers to fix incorrectly added information in a given test goal record. The reason for this approach is basic. A test goal itself does not change. It is either a valid test goal or it is not. If it is invalid, then it should be deleted. If it is valid, then it remains unchanged. To change a test goal's description is to change the test goal into a different test goal. A brief digression to explain the meaning of an invalid test goal is appropriate.

A test goal may be invalid for many reasons. First, it may simply have missed the mark of system requirements completely. Second, a test goal may be invalid because it was not specific enough, combining too much into one test goal. Testers may then delete the initial test goal and replace it with multiple succinct test goals (here, justification comments referring to the initial test goal may be appropriate for test history purposes). Third, a test goal may not be feasible in its current state or may require modification to conform better to the overall test plan. Again, justification comments would be appropriate.

The "Change" operation has an additional important function, namely to undelete previously deleted test goals. Should a previously deleted test goal need to be reinstated, then the "Change" operation

undeletes the test goal by marking the record's deleted flag as undeleted. The screen for test goal change screen input is very similar to the one for adding test goals, shown above. After the user inputs changes, a second screen allows the user to undelete all the test goal's previous links or else selectively revive test goal links to account for changes that may have occurred in the prototype's implementation code. TGTS reimplements the links for a user specified iteration. TGTS initiates the link undelete action by interactive questioning of the user.

c. Delete an Existing Test Goal

The third test goal operation is the "Delete" operation. This operation marks a test goal as deleted from the test plan. The test goal's record remains a part of the test goal database with a deleted field flagged. This operation is particularly valuable because it allows test goal information to be retained for record purposes. It also keeps records that may have to be added back in a subsequent iteration, should the results of an iteration demonstration dictate. Testers may review deleted test goals for a given iteration in determining what test plan modifications will be required in light of requirements changes. This operation is a bit more complex than previous ones in that it requires TGTS to delete all link records that mapped to the deleted test goal. TGTS deletes the links in the same way as the test goals so that they may be added back, if necessary. Before deleting a test goal, the goal history field should be annotated with the reason for deletion and a list of all replacement goals, if any. This maintains a trace of the test goal development. A sample test goal deletion screen is shown next.

```

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:
:          GOAL DELETION          :
:
:   Enter the goal number to delete:   40   :
:
:   Enter the iteration in which deleted:   3   :
:
:   Press 1 to cancel, 2 to exit,      :
:   and return to continue  20      :
:
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```

d. Link Test Goal to PSDL Operator/Data Stream

Test goals must be linked to the prototype implementation code, once developers add the functionality to the prototype. The "Link" operation links a test goal to an existing operator/data stream record. The "Link" module implementing this operation also works in the opposite direction, linking operator/data stream records to existing test goal records, making it one of the few modules in TGTS that is called by multiple modules. The "Link" operation explicitly allows the testers to link test goals to the prototype implementation code. A sample link database entry screen is shown below.

```

/*****
:                               LINK DATABASE DATA ENTRY                               :
*****/
GOAL NUMBER      0      OPERATOR/DATA STREAM NUMBER      0      DELETED      0

```

PSDL PART

ITERATION ADDED 0

```

/*****
: List the particular PSDL part by its
: grammatical title for a detailed location
: of a goal's implementation.
*****/

```

```

/*****
: to make corrections,
: Return to continue, C to cancel,
: or X to exit
*****/

```

e. Unlink Test Goal to PSDL Operator/Data Stream

"Unlink" allows links to be removed between a test goal record and an operator/data stream record. This allows the link database to be manipulated to account for changes in prototype implementation. Like the "Link" operation, the "Unlink" implementation is a shared operation with the operator/data stream database.

4. TGTS PSDL Update Operations

The PSDL (opstream) database is subject to five operations. Two of the operations, "Link" and "Unlink," are identical to those described above and are omitted here. The other three, "Add," "Delete" and "Change" are similar to the analogous operations for the test goals database, but have input screens to manipulate operator/data stream database records. The top-level menu screen for the PSDL database is shown next.

```

|MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:
:           PSDDL OPERATOR/DATA STREAM UPDATE MENU           :
:
:       Select your choice by number:                          :
:
:       1. Add a new operator/data stream                      :
:       2. Change an existing operator/data stream             :
:       3. Delete an existing operator/data stream             :
:       4. Link operator/data stream to test goal              :
:       5. Unlink operator/data stream to test goal            :
:       6. Return to Main Menu                                  :
:
:
:       Select your option:  0                                  :
:
:
|MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```

a. Add Operator/Data Stream

The "Add" operation, like its counterpart, allows implemented PSDL operators and data streams to be added to the database. There is an "Operator" boolean field in each record to flag each as either an operator or a data stream record since both PSDL parts reside in the same database. Each record automatically receives a unique operator number to distinguish each part and provide a key for linking PSDL parts with test goals. The "Add" input screen is shown next.


```

MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM;
:                                     ADD AN ITERATION HISTORY                               :
:                                                                              :
:                                     NEW ITERATION NUMBER   4                               :
:                                                                              :
:                                     NEW ITERATION HISTORY  memo                          :
:                                     Cntl+PgDn to enter, Cntl+PgUp to exit                 :
:DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD:
:                                     Enter 1 to cancel, 2 to accept entry   2             :
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM<

```

b. Change an Existing Iteration

The "Change" operation allows the user to change any data field. The user should be careful about changing the iteration number field, since this is a key field. The input screen is very similar to the one for adding records.

c. Delete an Existing Iteration

This operation allows the user to delete an iteration record permanently. This operation should be used with care to avoid inadvertent loss of data. The program asks for the iteration to delete, retrieves it and shows it to the user. The program then asks the user to confirm the deletion. The default value is "No." TGTS then takes the requested action. This operation is useful for removing records that require massive corrections or for removing erroneous entries.

d. List All Iteration Information

The listing operations for iteration information are separate from all the other output operations in TGTS. The "List All" operation lists all the iteration information records in iteration order. The user may specify output to the screen only or to the screen and printer.

e. List Information for Iteration X

The "List for Iteration X" operation allows the user to specify a single iteration for output. The user may specify output to the screen only or to the screen and printer.

f. List Most Recent Iteration Number

The "List Most Recent Iteration Number" operation simply checks the Iteration Information database for the last iteration number and returns it. Provided that users add an iteration information record with every iteration, the operation gives last iteration of development and testing. The operation provides a reminder to the user of the last iteration recorded.

6. Exiting TGTS

Users exit TGTS very simply. The user returns to the top-level TGTS menu and selects the "Quit" option. The "Quit" option saves all files and closes all files, then exits TGTS. A "Good Bye for Now" screen appears with the dBase command line dot prompt at the bottom. The user types in "QUIT" followed by a carriage return, and the dBase program concludes execution, completing the process.

C. TGTS USAGE

TGTS is a flexible tool. A particular test team's requirements for recording testing information can easily be fit into the text recording fields available. The information stored in the other fields is either standard within requirements-based testing (i.e., test class) or else is standard for CAPS (i.e., PSDL grammar). Almost any test goal information can therefore be stored in TGTS.

Chapter IV of the thesis, and Appendix A combine to provide a simple example of TGTS usage. There are several TGTS user techniques that bear particular mention and vary in nature from administrative to technical.

1. Memo Fields

All the text fields in the records are dBase memo fields. To keep records vertically separated for output, always include a couple of carriage returns at the conclusion of text in the fields. DBase makes no provisions for separating fields vertically.

While not mandating any exact formats for filling text fields in TGTS, entries should be reasonably consistent in form and wording. The more organized the users keep the text fields, the more useful TGTS will be.

2. Input Guards

TGTS guards most data field input with dBase constructs that restrict the type, size and range of data. TGTS enforces the types, sizes and ranges but the user must ensure the data's correctness. TGTS simply stores all text field data and does not analyze it in any way. TGTS checks key fields for consistency and currency, especially in linking operations.

3. Deletion Operations

When test or prototype changes require record deletions, users should ensure that they first update the history fields since they can only be accessed while records are flagged as undeleted. If a user fails to update a history prior to record deletion, then the record must first be undeleted, then updated, then deleted again.

Based upon present experience with TGTS, the following sequence seems best for test goal and operator/data stream deletion. The method described is for test goal deletion but applies in mirror-fashion for operator/data stream deletion.

1. Annotate the test goal history field with the reason and iteration of deletion. Note any test goal numbers that are replacing the one to be deleted.
2. Note all links from the test goal to be deleted. Analyze the old links appropriately for signifying new test goal links.
3. Add any replacement test goals, noting the test goals they replace or decompose in the test goal history fields of the new test goals.
4. Add new links, as appropriate.
5. Delete the old test goal.

D. TGTS SPECIAL FEATURES AND LIMITATIONS

TGTS is a research tool designed to demonstrate the feasibility of testing tools to support evolutionary iterative rapid prototyping. It provides the essential functionality to link test goals to their implementation in the prototype's source code and to provide the tester with output in many forms useful for test planning. TGTS greatest strengths are its requirements-based test goal information recording capability, its linking capability and its multiple data output formats.

As a demonstration tool, not integrated into the CAPS environment, TGTS has very limited consistency checking. It does check link consistency and its output reports can be used in cross referencing to determine if all data is consistent. Further, the design of the data

output reports makes manual consistency checking easier than with the traditional, manual testing methods.

TGTS will be as good as the data entry and the formatting of data in the free text fields (dBase memo fields). If users use consistent text entry formats, then consistency checking by review will be much easier. With time, users can determine the best formatting methods to use.

TGTS has only one set of database files and no way to archive them, therefore TGTS can only be used for one prototype at a time. Should users need to start another project, they must first rename all the existing database files (four), and then recreate the (new) database files with the same names (GOALS.DBF, OPSTREAM.DBF, RO_LINK.DBF and ITERATNS.DBF) by using the dBase "CREATE" command. Index files do not need to be archived or deleted. The "INDEX_IT" utility (next paragraph) regenerates correct index files for the above four database files.

If users wish to access the TGTS database files directly with dBase, there is no safeguard to prevent this. At the current stage of research, this is beneficial because it allows easy manipulation of the databases with the dBase EDIT and BROWSE commands. After modifying data files, they can be indexed by calling the utility program "INDEX_IT.PRG" with the command "DO INDEX_IT." This utility program reindexes all the index files for all the databases.

The dBase screen files TGTS uses for data entry allow the user to cycle through a record and into the next for certain screen entries, adding a blank or incomplete record without the user meaning to do so. The user should be careful before entering the last record entry field to prevent adding useless records. As long as the user is in the entry

screen for a record, the cursor can be moved back to previous fields using the up and down arrows. Once the user enters the last field, TGTS adds the record. The record may be completed or corrected simply by performing a "Change" operation. The only way to remove inadvertent records permanently is to use dBase editing commands. While removal of erroneously added records is not essential (just ensure they contain no information and mark them deleted), it keeps the database uncluttered by removing them.

TGTS is intended for further research and will hopefully be extended and improved over time. It has been kept simple to provide an easy to use requirements-based testing tool for CAPS. The source code for TGTS is in Appendix C.

APPENDIX C TGTS SOURCE CODE

```

*****
*   Program Name   : TESTGOAL.PRG
*   Author        : Ned Davis
*   Date          : 16 Aug 90
*   Revised       :
*   Language      : Dbase III+
*   Description    : Displays a welcome screen to the TESTGOAL
*                   System. A menu is next presented for functional
*                   options selection. Appropriate subprograms are then
*                   called. THIS IS THE MASTER PROGRAM.
*****

**** set environment ****
SET TALK OFF
SET SCOREBOARD OFF
SET STATUS OFF
SET MEMOWIDTH TO 30
choice = 0

**** display welcome screen ****
DO WELCOME

**** display main menu ****
DO WHILE choice #5
  choice = 0
  SET COLOR TO R+/N,BG/N,GR+
  CLEAR
  @ 3,10 TO 20,70 DOUBLE
  SET COLOR TO GR+/
  @ 5,32 SAY "M A I N M E N U"
  SET COLOR TO G/
  @ 7,26 SAY "Select your choice by number: "
  @ 9,20 SAY "1. Database outputs: screen and hard copy"
  @ 10,20 SAY "2. Test goal updates"
  @ 11,20 SAY "3. PSDL operator/data stream updates"
  @ 12,20 SAY "4. Prototype iteration information"
  @ 13,20 SAY "5. Quit"
  SET COLOR TO W+/
  @ 16,26 SAY "Enter your choice here: " GET choice PICTURE "99";
  RANGE 0,5
  READ

  **** perform user's request ****
  DO CASE

```

```
        CASE choice=1
            DO REPORTS
        CASE choice=2
            DO UP_GOALS
        CASE choice=3
            DO UP_PSDL
        CASE choice = 4
            DO ITERINFO
    ENDCASE

ENDDO
CLEAR

**** display bye screen ****
DO BYE
```

```
* Program Name : WELCOME.PRG
* Author       : Ned Davis
* Date        : 16 Aug 90
* Revised     :
* Language    : Dbase III+
* Description  : Displays a welcome screen that briefly
*               describes the TESTGOAL system.
*****
```

SET COLOR TO BG+/B,GR+/R,G

CLEAR

@ 6,20 SAY "WELCOME TO THE CAPS TEST GOAL TRACKING SYSTEM"

@ 8,12 SAY ""

TEXT

This is a requirements-based test goal tracking system (TCTS) designed to accompany prototype development in CAPS. As verifiable test goals are determined, they are added to the database. When the behavior is implemented in CAPS, the PSDL code implementing the testable behavior is associated with the proper test goal in the database for ease of reference during the development and testing of the prototype. Test goals and PSDL operators and data streams can be tracked to note changes between prototype iterations. System capabilities include data add, delete, change, annotation condition search and multiple report form outputs.

ENDTEXT

@ 22,1 SAY ""

SET COLOR TO R+/B

@ 5,10 to 22,70 DOUBLE

SET COLOR TO W+/B

WAIT

```

*****
* Program Name : REPORTS.PRG
* Author      : Ned Davis
* Date       : 1 Oct 90
* Revised    :
* Language   : dBase III+
* Description : Provides a menu screen for allowing report
*              output of the type selected. Called by
*              TESTGOAL.PRG.
*****

```

```

**** set environment ****

```

```

choice = 0

```

```

**** display menu ****

```

```

DO WHILE choice # 17

```

```

    choice = 0

```

```

    print_it = .F.    && flag to send output to printer

```

```

    SET COLOR TO R/BG,GR+/R,G

```

```

    CLEAR

```

```

    SET COLOR TO GR+/BG

```

```

    @ 2,32 SAY "REPORT OUTPUT MENU"

```

```

    SET COLOR TO R/BG

```

```

    @ 3, 0 TO 24,79 DOUBLE

```

```

    @ 14, 1 TO 14,78 DOUBLE

```

```

    @ 18, 1 TO 18,78 DOUBLE

```

```

    @ 20, 1 TO 20,78 DOUBLE

```

```

    @ 4,40 TO 13,40 DOUBLE

```

```

    SET COLOR TO GR+/BG

```

```

    @ 4, 7 SAY "TEST GOAL OUTPUT OPTIONS"

```

```

    @ 4,43 SAY "OPERATOR/DATA STREAM OUTPUT OPTIONS"

```

```

    @ 6, 2 SAY "1. All test goals"

```

```

    @ 6,42 SAY "9. All operators"

```

```

    @ 7, 2 SAY "2. Test goals added in iteration x"

```

```

    @ 7,42 SAY "10. All data streams"

```

```

    @ 8, 2 SAY "3. Test goals modified in iteration x"

```

```

    @ 8,42 SAY "11. Op/DStrm added in iteration x"

```

```

    @ 9, 2 SAY "4. Test goals deleted in iteration x"

```

```

    @ 9,42 SAY "12. Op/DStrm modified in iteration x"

```

```

    @ 10, 2 SAY "5. Test goals that map to an Op/DStrm"

```

```

    @ 10,42 SAY "13. Op/DStrm deleted in iteration x"

```

```

    @ 11, 2 SAY "6. Test goals of aggregate x"

```

```

    @ 11,42 SAY "14. Op/DStrms that map to a test goal"

```

```

    @ 12, 2 SAY "7. Test goals of priority x"

```

```

    @ 13, 2 SAY "8. Test goals of class x"

```

```

    @ 15,30 SAY "LINK OUTPUT OPTIONS"

```

```

    @ 17, 2 SAY "15. Global Link outputs"

```

```

    @ 17,42 SAY "16. Iteration Link outputs"

```

```

    @ 19,28 SAY "17. Return to Main Menu"

```

```

    SET COLOR TO W+/BG

```

```

    @ 21,26 SAY "Enter your choice here: " GET choice PICTURE "999";

```

```

        RANGE 0,17
READ
IF choice # 17
    @ 22,22 SAY "Send output to printer (Y or N)? :" GET print_it ;
        PICTURE "Y"
    READ
ENDIF
CLEAR

**** perform user's request ****
DO CASE
    CASE choice = 1
        DO R_CASE1
    CASE choice = 2
        DO R_CASE2
    CASE choice = 3
        DO R_CASE3
    CASE choice = 4
        DO R_CASE4
    CASE choice = 5
        DO R_CASE5
    CASE choice = 6
        DO R_CASE6
    CASE choice = 7
        DO R_CASE7
    CASE choice = 8
        DO R_CASE8
    CASE choice = 9
        DO R_CASE9
    CASE choice = 10
        DO R_CASE10
    CASE choice = 11
        DO R_CASE11
    CASE choice = 12
        DO R_CASE12
    CASE choice = 13
        DO R_CASE13
    CASE choice = 14
        DO R_CASE14
    CASE choice = 15
        DO R_CASE15
    CASE choice = 16
        DO R_CASE16
ENDCASE
ENDDO

**** restore environment ****
CLEAR
choice = 0
RETURN TO MASTER

```

```
* Program Name : R_CASE1.PRG
* Author       : Ned Davis
* Date        : 1 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for test goal output.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

choicel = 0

choicela = 1

SET MEMOWIDTH TO 25

**** display menu ****

DO WHILE choicel # 6

choicel = 0

SET COLOR TO R/BG,GR+/R,G

CLEAR

@ 3,10 TO 23,70 DOUBLE

@ 18,11 TO 18,69 DOUBLE

SET COLOR TO GR+/BG

@ 5,30 SAY "TEST GOAL LISTING OPTIONS"

@ 7,20 SAY "Select your choice by number: "

@ 9,20 SAY "1. List by Test Goal Number"

@ 10,20 SAY "2. List by Testing Priority"

@ 11,20 SAY "3. List by Test Aggregate"

@ 12,20 SAY "4. List by Test Class"

@ 13,20 SAY "5. List deleted goals"

@ 14,20 SAY "6. Return to Report Menu"

SET COLOR TO W+/BG

@ 17,15 SAY "Enter your choice here: " GET choicel PICTURE "99";
RANGE 0,6

READ

SET COLOR TO GR+/BG

IF choicel # 6

choicela = 1

@ 19,15 SAY "1. Brief output: description & history omitted"

@ 20,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choicela ;
PICTURE "99" RANGE 1,2

READ

SET COLOR TO GR+/BG

ENDIF

**** perform choices ****

```

DO CASE
CASE choicel = 1
  USE goals INDEX goal_num
  IF choicela = 1
    IF print_it
      LIST FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL FOR deleted = 0 OFF
      WAIT
    ENDIF
  ELSE
    SET MEMOWIDTH TO 30
    IF print_it
      LIST goal_num, goal_descr, g_history;
      FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL goal_num, goal_descr, g_history;
      FOR deleted = 0 OFF
      WAIT
    ENDIF
    SET MEMOWIDTH TO 25
  ENDIF
CASE choicel = 2
  USE goals INDEX test_pri
  IF choicela = 1
    IF print_it
      LIST FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL FOR deleted = 0 OFF
      WAIT
    ENDIF
  ELSE
    IF print_it
      LIST test_pri, goal_num, goal_descr, g_history;
      FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL test_pri, goal_num, goal_descr, g_history;
      FOR deleted = 0 OFF
      WAIT
    ENDIF
  ENDIF
CASE choicel = 3
  USE goals INDEX aggregat
  IF choicela = 1
    IF print_it
      LIST FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL FOR deleted = 0 OFF
      WAIT
    ENDIF
  ELSE

```

```

IF print_it
  LIST aggregate, goal_num, goal_descr, g_history;
  FOR deleted = 0 OFF TO PRINT
ELSE
  DISPLAY ALL aggregate, goal_num, goal_descr, g_history;
  FOR deleted = 0 OFF
  WAIT
ENDIF
ENDIF
CASE choicel = 4
  USE goals INDEX testclas
  IF choicela = 1
    IF print_it
      LIST FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL FOR deleted = 0 OFF
      WAIT
    ENDIF
  ELSE
    IF print_it
      LIST test_class, goal_num, goal_descr, g_history;
      FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL test_class, goal_num, goal_descr, g_history;
      FOR deleted = 0 OFF
      WAIT
    ENDIF
  ENDIF
CASE choicel = 5
  DO DEL_GOALS
ENDCASE
CLOSE DATABASES
ENDDO

CLEAR
SET MEMOWIDTH TO 30
RETURN

```

```
* Program Name   : DEL_GOLS.PRG
* Author        : Ned Davis
* Date          : 7 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Allows deleted test goals to be listed
*                  to screen or printer in either brief or
*                  descriptive format. Called by R_CASE1.PRG.
*                  WARNING - uses global variable print_it
*                  from REPORTS.PRG and choicela from R_CASE1.PRG.
*****
```

**** set memory variables ****

del_ch = 0

iter = 0

**** display menu ****

DO WHILE del_ch # 3

del_ch = 0

iter = 0

CLEAR

SET COLOR TO R/BG

@ 3,10 TO 16,70 DOUBLE

SET COLOR TO GR+/BG

@ 4,25 SAY "DELETED TEST GOALS LISTING OPTIONS"

@ 6,20 SAY "Select your choice by number: "

@ 8,20 SAY "1. List all deleted test goals"

@ 9,20 SAY "2. List test goals deleted in an iteration"

@ 10,20 SAY "3. Return to Test Goal Menu"

SET COLOR TO W+/BG

@ 12,20 SAY "Enter your choice here: " GET del_ch;

PICTURE "99" RANGE 1,3

READ

SET COLOR TO GR+/BG

DO CASE

CASE del_ch = 1 && list all deleted test goals

USE goals INDEX goal_num

IF choicela = 1

IF print_it

LIST FOR deleted # 0 OFF TO PRINT

ELSE

DISPLAY ALL FOR deleted #0 OFF

WAIT

ENDIF

ELSE

IF print_it

LIST goal_num, goal_descr, g_history FOR deleted # 0;

OFF TO PRINT

ELSE

DISPLAY ALL goal_num, goal_descr, g_history;

```

        FOR deleted # 0 OFF
        WAIT
    ENDIF
ENDIF
CASE del_ch = 2    && list deleted test goals for iter x
    SET COLOR TO W+/BG
    @ 14,20 SAY "Enter the iteration to output: ";
        GET iter PICTURE "999" RANGE 1,99
    READ
    SET COLOR TO GR+/BG
    USE goals INDEX delet_tg
    SEEK iter
    IF FOUND()
        IF choicela = 1
            IF print_it
                LIST WHILE deleted = iter OFF TO PRINT
            ELSE
                DISPLAY WHILE deleted = iter OFF
                WAIT
            ENDIF
        ELSE
            IF print_it
                SET PRINT ON
                ? "TEST GOALS DELETED FOR ITERATION ",iter
                SET PRINT OFF
                LIST goal_num, goal_descr, g_history;
                    WHILE deleted = iter OFF TO PRINT
            ELSE
                ? "TEST GOALS DELETED FOR ITERATION ",iter
                DISPLAY goal_num, goal_descr, g_history;
                    WHILE deleted = iter OFF
                WAIT
            ENDIF
        ENDIF
    ELSE
        CLEAR
        @ 5,5
        ? "No deleted test goals found for iteration ",iter
        WAIT
    ENDIF
ENDCASE    && return to caller
CLOSE DATABASES
ENDDO
RETURN

```

```

*****
*   Program Name   : R_CASE2.PRG
*   Author        : Ned Davis
*   Date          : 3 Oct 90
*   Revised       :
*   Language      : Dbase III+
*   Description    : Provides menu options for test goal output.
*                   Outputs test goals added in a given iteration.
*                   Sends output to screen or printer as req'd.
*                   Called by REPORTS.PRG. !!WARNING- uses global
*                   variable "print_it" from calling program to
*                   det'm where to send output.
*****

```

```

**** set environment ****
choice2 = 1
iter_no = 0

```

```

**** display menu ****
SET COLOR TO R/BG,GR+/R,G
CLEAR
@ 3,10 TO 17,70 DOUBLE
@ 9,11 TO 9,69 DOUBLE
@ 13,11 TO 13,69 DOUBLE
SET COLOR TO GR+/BG
@ 5,24 SAY "TEST GOALS ADDED FOR A GIVEN ITERATION"
@ 7,20 SAY "Enter the iteration to output or 0 to escape"
SET COLOR TO W+/BG
@ 11,15 SAY "Enter your choice here: " GET iter_no PICTURE "999";
    RANGE 0,99

```

```

READ
SET COLOR TO GR+/BG
IF iter_no # 0
    choice2 = 1
    @ 14,15 SAY "1. Brief output: description & history omitted"
    @ 15,15 SAY "2. Descriptive output: w/ description & history"
    SET COLOR TO W+/BG
    @ 21,15 SAY "Enter your choice here: " GET choice2 ;
        PICTURE "99" RANGE 1,2

```

```

READ
SET COLOR TO GR+/BG

```

```

**** perform choices ****
USE goals INDEX iter_add
SEEK iter_no
IF FOUND()
    IF choice2 = 1    && brief format
        IF print_it
            LIST WHILE iter_added = iter_no OFF TO PRINT
        ELSE
            DISPLAY WHILE iter_added = iter_no OFF

```

```

        WAIT
    ENDIF
ELSEF                                     && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS ADDED IN ITERATION ",iter_no
        SET PRINT OFF
        LIST goal_num, goal_descr, g_history ;
        WHILE iter_added = iter_no OFF TO PRINT
    ELSE
        ? "TEST GOALS ADDED IN ITERATION ",iter_no
        DISPLAY goal_num, goal_descr, g_history ;
        WHILE iter_added = iter_no OFF
    WAIT
    ENDIF
ENDIF
ELSE                                     && iteration not found
    CLEAR
    @ 5,5
    ? "No test goals found for iteration ",iter_no
    ?? CHR(7)
    WAIT
ENDIF
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE3.PRG
* Author      : Ned Davis
* Date       : 3 Oct 90
* Revised    :
* Language   : Dbase III+
* Description : Provides menu options for test goal output.
*              Outputs test goals modified in a given iteration.
*              Sends output to screen or printer as req'd.
*              Called by REPORTS.PRG. !!WARNING- uses global
*              variable "print_it" from calling program to
*              det'm where to send output.
```

**** set environment ****

```
choice2 = 1
iter_no = 0
```

**** display menu ****

```
SET COLOR TO R/BG,GR+/R,G
CLEAR
@ 3,10 TO 17,70 DOUBLE
@ 9,11 TO 9,69 DOUBLE
@ 13,11 TO 13,69 DOUBLE
SET COLOR TO GR+/BG
@ 5,20 SAY "TEST GOALS MODIFIED FOR A GIVEN ITERATION"
@ 7,20 SAY "Enter the iteration to output or 0 to escape"
SET COLOR TO W+/BG
@ 11,15 SAY "Enter your choice here: " GET iter_no PICTURE "999";
    RANGE 0,99
```

READ

SET COLOR TO GR+/BG

IF iter_no # 0

choice2 = 1

@ 14,15 SAY "1. Brief output: description & history omitted"

@ 15,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choice2 ;

PICTURE "99" RANGE 1,2

READ

SET COLOR TO GR+/BG

**** perform choices ****

USE goals INDEX updated

SEEK iter_no

IF FOUND()

IF choice2 = 1 && brief format

IF print_it

LIST WHILE updated = iter_no OFF TO PRINT

ELSE

DISPLAY WHILE updated = iter_no OFF

```

        WAIT
    ENDIF
ELSE
    && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS MODIFIED IN ITERATION ",iter_no
        SET PRINT OFF
        LIST goal_num, goal_descr, g_history ;
        WHILE updated = iter_no OFF TO PRINT
    ELSE
        ? "TEST GOALS MODIFIED IN ITERATION ",iter_no
        DISPLAY goal_num, goal_descr, g_history ;
        WHILE updated = iter_no OFF
    WAIT
    ENDIF
ENDIF
ELSE
    && iteration not found
    CLEAR
    @ 5,5
    ? "No modified test goals found for iteration ",iter_no
    ?? CHR(7)
    WAIT
ENDIF
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE4.PRG
* Author       : Ned Davis
* Date        : 3 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for test goal output.
*               Outputs test goals deleted in a given iteration.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

```
choice2 = 1
iter_no = 0
```

**** display menu ****

```
SET COLOR TO R/BG,GR+/R,G
CLEAR
@ 3,10 TO 17,70 DOUBLE
@ 9,11 TO 9,69 DOUBLE
@ 13,11 TO 13,69 DOUBLE
SET COLOR TO GR+/BG
@ 5,20 SAY "TEST GOALS DELETED FOR A GIVEN ITERATION"
@ 7,18 SAY "Enter the iteration to output or 0 to escape"
SET COLOR TO W+/BG
@ 11,15 SAY "Enter your choice here: " GET iter_no PICTURE "999";
    RANGE 0,99
```

READ

SET COLOR TO GR+/BG

IF iter_no # 0

choice2 = 1

@ 14,15 SAY "1. Brief output: description & history omitted"

@ 15,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choice2 ;

PICTURE "99" RANGE 1,2

READ

SET COLOR TO GR+/BG

**** perform choices ****

USE goals INDEX delet_tg

SEEK iter_no

IF FOUND()

IF choice2 = 1 && brief format

IF print_it

LIST WHILE deleted = iter_no OFF TO PRINT

ELSE

DISPLAY WHILE deleted = iter_no OFF

```

        WAIT
    ENDIF
ELSE
    && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS DELETED IN ITERATION ",iter_no
        SET PRINT OFF
        LIST goal_num, goal_descr, g_history ;
        WHILE deleted = iter_no OFF TO PRINT
    ELSE
        ? "TEST GOALS DELETED IN ITERATION ",iter_no
        DISPLAY goal_num, goal_descr, g_history ;
        WHILE deleted = iter_no OFF
    WAIT
    ENDIF
ENDIF
ELSE
    && iteration not found
    CLEAR
    @ 5,5
    ? "No deleted test goals found for iteration ",iter_no
    ?? CHR(7)
    WAIT
ENDIF
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE5.PRG
* Author       : Ned Davis
* Date        : 7 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for test goal output.
*               Outputs test goals that map to a particular
*               operator/data stream - either all or for an
*               iteration.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

```
choice5 = 0
iter_no = 0
choicela = 0
opstm = 0
```

**** display menu ****

```
DO WHILE choice5 # 3
  choice5 = 0
  iter_no = 0
  choicela = 0
  opstm = 0
  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 23,70 DOUBLE
  @ 14,11 TO 14,69 DOUBLE
  @ 17,11 TO 17,69 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,18 SAY "TEST GOALS THAT MAP TO A GIVEN OP/DATA STREAM"
  @ 7,20 SAY "Select your choice by number: "
  @ 9,20 SAY "1. List Test Goals for all iterations"
  @ 10,20 SAY "2. List Test Goals for a given iteration"
  @ 11,20 SAY "3. Return to Report Menu"
  SET COLOR TO W+/BG
  @ 13,20 SAY "Enter your choice here: " GET choice5;
    PICTURE "99" RANGE 0,3
  READ
  SET COLOR TO GR+/BG
  IF choice5 = 2
    @ 15,20 SAY "Enter the iteration to output or 0 to escape"
    SET COLOR TO W+/BG
    @ 16,20 SAY "Enter your choice here: " GET iter_no PICTURE "999";
      RANGE 0,99
    READ
    SET COLOR TO GR+/BG
```

```

ENDIF
IF iter_no # 0 .OR. choice5 # 3
    choicela = 1
    @ 18,15 SAY "1. Brief output: description & history omitted"
    @ 19,15 SAY "2. Descriptive output: w/ description & history"
    SET COLOR TO W+/BG
    @ 21,15 SAY "Enter your choice here: " GET choicela ;
        PICTURE "99" RANGE 1,2
    READ
    @ 22,15 SAY "Enter the op/data stream no. for goal listing: ";
        GET opstm PICTURE "99999" RANGE 1,9999
    READ
    SET COLOR TO GR+/BG
ENDIF

**** perform choices ****
CLEAR
DO CASE
    CASE choice5 = 1
        DO TRACE1OP
    CASE choice5 = 2
        DO TRACE2OP
ENDCASE
CLOSE DATABASES
ENDDO
CLEAR
RETURN

```

```

*****
* Program Name   : TRACE10P.PRG
* Author        : Ned Davis
* Date          : 8 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Prints to screen or printer a listing of
*                  test goals that link to a particular op/
*                  data stream. Called from R_CASE5.PRG.
*                  WARNING!! - uses vars from R_CASE5.PRG.
*****

```

```

**** set work areas ****

```

```

SELECT A
USE ro_link INDEX onumlink

```

```

SELECT B
USE goals INDEX goal_num

```

```

**** set up relationships ****
SELECT A
SET RELATION TO goal_num INTO goals

```

```

**** output report ****

```

```

GO TOP
SEEK opstm
IF POUND()
  IF choicela = 1          && brief format
    IF print_it
      SET PRINT ON
      ? "TEST GOALS THAT MAP TO OP/DATA STREAM ",opstm
      SET PRINT OFF
      DO WHILE op_number = opstm
        SELECT B
        LIST FOR goal_num = A->goal_num OFF TO PRINT
        SELECT A
        SKIP
      ENDDO
    ELSE
      ? "TEST GOALS THAT MAP TO OP/DATA STREAM",opstm
      DO WHILE op_number = opstm
        SELECT B
        DISPLAY FOR goal_num = A->goal_num OFF
        SELECT A
        SKIP
      ENDDO
    WAIT
  ENDIF
ELSE
  && detailed format
  IF print_it
    SET PRINT ON

```

```

? "TEST GOALS THAT MAP TO OP/DATA STREAM ",opstm
SET PRINT OFF
DO WHILE op_number = opstm
    SELECT B
        LIST goal_num, goal_descr, g_history;
        FOR goal_num = A->goal_num OFF TO PRINT
    SELECT A
    SKIP
ENDDO
ELSE
? "TEST GOALS THAT MAP TO OP/DATA STREAM ",opstm
DO WHILE op_number = opstm
    SELECT B
        DISPLAY goal_num, goal_descr, g_history;
        FOR goal_num = A->goal_num OFF
    SELECT A
    SKIP
ENDDO
WAIT
ENDIF
ENDIF
ELSE
                                && opnum not found
    CLEAR
    @ 5,5
    ? "No operator/data stream found to match ",opstm
    WAIT
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name   : TRACE20P.PRG
* Author        : Ned Davis
* Date          : 8 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Prints to screen or printer a listing of
*                  test goals that link to a particular op/
*                  data stream, where the links were added in a
*                  given iteration. Called from R_CASE5.PRG.
*                  WARNING!! - uses vars from R_CASE5.PRG.
*****
```

**** set work areas ****

```
SELECT A
USE ro_link INDEX onumlink
```

```
SELECT B
USE goals INDEX goal_num
```

**** set up relationships ****

```
SELECT A
SET RELATION TO goal_num INTO goals
```

**** output report ****

```
GO TOP
SEEK opstm
IF FOUND()
  IF choicela = 1      && brief format
    IF print_it
      SET PRINT ON
      ? "TEST GOALS THAT MAP TO OP/DATA STREAM ",opstm
      ?? "  FOR ITERATION ",iter_no
      SET PRINT OFF
      DO WHILE op_number = opstm
        IF l_iteradd = iter_no
          SELECT B
          LIST FOR goal_num = A->goal_num OFF TO PRINT
          SELECT A
        ENDIF
        SKIP
      ENDDO
    ELSE
      ? "TEST GOALS THAT MAP TO OP/DATA STREAM",opstm
      ?? "  FOR ITERATION ",iter_no
      DO WHILE op_number = opstm
        IF l_iteradd = iter_no
          SELECT B
          DISPLAY FOR goal_num = A->goal_num OFF
          SELECT A
        ENDIF
```

```

        SKIP
    ENDF
    WAIT
ENDIF
ELSE
    && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS THAT MAP TO OP/DATA STREAM ",opstm
        ?? " FOR ITERATION ", iter_no
        SET PRINT OFF
        DO WHILE op_number = opstm
            IF 1_iteradd = iter_no
                SELECT B
                LIST goal_num, goal_descr, g_history;
                FOR goal_num = A->goal_num OFF TO PRINT
                SELECT A
            ENDIF
            SKIP
        ENDDO
    ELSE
        ? "TEST GOALS THAT MAP TO OP/DATA STREAM ",opstm
        ?? " FOR ITERATION ", iter_no
        DO WHILE op_number = opstm
            IF 1_iteradd = iter_no
                SELECT B
                DISPLAY goal_num, goal_descr, g_history;
                FOR goal_num = A->goal_num OFF
                SELECT A
            ENDIF
            SKIP
        ENDDO
    WAIT
ENDIF
ENDIF
ELSE
    && opnum not found
    CLEAR
    @ 5,5
    ? "No operator/data stream found to match ",opstm
    ? " for iteration ",iter_no
    WAIT
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE6.PRG
* Author       : Ned Davis
* Date        : 3 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for test goal output.
*               Outputs test goals of a given test aggregate.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

choice2 = 1

agg = 0

**** display menu ****

SFT COLOR TO R/BG,GR+/R,G

CLEAR

@ 3,10 TO 17,70 DOUBLE

@ 9,11 TO 9,69 DOUBLE

@ 13,11 TO 13,69 DOUBLE

SET COLOR TO GR+/BG

@ 5,24 SAY "TEST GOALS IN A GIVEN TEST AGGREGATE"

@ 7,20 SAY "Enter the aggregate to output or 0 to escape"

SET COLOR TO W+/BG

@ 11,15 SAY "Enter your choice here: " GET agg PICTURE "99";

RANGE 0,99

READ

SET COLOR TO GR+/BG

IF agg # 0

choice2 = 1

@ 14,15 SAY "1. Brief output: description & history omitted"

@ 15,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choice2 ;

PICTURE "99" RANGE 1,2

READ

SET COLOR TO GR+/BG

**** perform choices ****

USE goals INDEX aggregat

SEEK agg

IF FOUND()

IF choice2 = 1 && brief format

IF print_it

LIST WHILE aggregate = agg OFF TO PRINT

ELSE

DISPLAY WHILE aggregate = agg OFF

```

        WAIT
    ENDIF
ELSE
    && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS IN AGGREGATE ",agg
        SET PRINT OFF
        LIST goal_num, goal_descr, g_history ;
        WHILE aggregate = agg OFF TO PRINT
    ELSE
        ? "TEST GOALS IN AGGREGATE ",agg
        DISPLAY goal_num, goal_descr, g_history ;
        WHILE aggregate = agg OFF
    WAIT
    ENDIF
ENDIF
ELSE
    && iteration not found
    CLEAR
    @ 5,5
    ? "No test goals found for aggregate ",agg
    ?? CHR(7)
    WAIT
ENDIF
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE7.PRG
* Author       : Ned Davis
* Date        : 3 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for test goal output.
*               Outputs test goals of priority x.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

```
choice2 = 1
priority = 0
```

**** display menu ****

```
SET COLOR TO R/BG,GR+/R,G
CLEAR
@ 3,10 TO 17,70 DOUBLE
@ 9,11 TO 9,69 DOUBLE
@ 13,11 TO 13,69 DOUBLE
SET COLOR TO GR+/BG
@ 5,24 SAY "TEST GOALS OF A GIVEN PRIORITY"
@ 7,20 SAY "Enter the priority to output or 0 to escape"
SET COLOR TO W+/BG
@ 11,15 SAY "Enter your choice here: " GET priority PICTURE "99";
    RANGE 0,9
```

READ

SET COLOR TO GR+/BG

IF priority # 0

choice2 = 1

@ 14,15 SAY "1. Brief output: description & history omitted"

@ 15,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choice2 ;

PICTURE "99" RANGE 1,2

READ

SET COLOR TO GR+/BG

**** perform choices ****

USE goals INDEX test_pri

SEEK priority

IF FOUND()

IF choice2 = 1 && brief format

IF print_it

LIST WHILE test_pri = priority OFF TO PRINT

ELSE

DISPLAY WHILE test_pri = priority OFF

```

        WAIT
    ENDIF
ELSE
    && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS OF PRIORITY ",priority
        SET PRINT OFF
        LIST goal_num, goal_descr, g_history ;
        WHILE test_pri = priority OFF TO PRINT
    ELSE
        ? "TEST GOALS OF PRIORITY ",priority
        DISPLAY goal_num, goal_descr, g_history ;
        WHILE test_pri = priority OFF
    WAIT
    ENDIF
ENDIF
ELSE
    && iteration not found
    CLEAR
    @ 5,5
    ? "No test goals found for priority ",priority
    ?? CHR(7)
    WAIT
ENDIF
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```

*****
* Program Name : R_CASE8.PRG
* Author      : Ned Davis
* Date       : 3 Oct 90
* Revised    :
* Language   : Dbase III+
* Description : Provides menu options for test goal output.
*              Outputs test goals of class x.
*              Sends output to screen or printer as req'd.
*              Called by REPORTS.PRG. !!WARNING- uses global
*              variable "print_it" from calling program to
*              det'm where to send output.
*****

```

```

**** set environment ****

```

```

choice2 = 1
tclass = 0

```

```

**** display menu ****

```

```

SET COLOR TO R/BG,GR+/R,G
CLEAR
@ 3,10 TO 17,70 DOUBLE
@ 9,11 TO 9,69 DOUBLE
@ 13,11 TO 13,69 DOUBLE
SET COLOR TO GR+/BG
@ 5,24 SAY "TEST GOALS OF A GIVEN CLASS"
@ 7,20 SAY "Enter the class to output or 0 to escape"
SET COLOR TO W+/BG
@ 11,15 SAY "Enter your choice here: " GET tclass PICTURE "99";
    RANGE 0,4

```

```

READ

```

```

SET COLOR TO GR+/BG
IF tclass # 0
    choice2 = 1
    @ 14,15 SAY "1. Brief output: description & history omitted"
    @ 15,15 SAY "2. Descriptive output: w/ description & history"
    SET COLOR TO W+/BG
    @ 21,15 SAY "Enter your choice here: " GET choice2 ;
        PICTURE "99" RANGE 1,2

```

```

READ

```

```

SET COLOR TO GR+/BG

```

```

**** perform choices ****

```

```

USE goals INDEX testclas
SEEK tclass
IF FOUND()
    IF choice2 = 1    && brief format
        IF print_it
            LIST WHILE test_class = tclass OFF TO PRINT
        ELSE
            DISPLAY WHILE test_class = tclass OFF

```

```

        WAIT
    ENDIF
ELSE
    && detailed format
    IF print_it
        SET PRINT ON
        ? "TEST GOALS IN TEST CLASS ",tclass
        SET PRINT OFF
        LIST goal_num, goal_descr, g_history ;
        WHILE test_class = tclass OFF TO PRINT
    ELSE
        ? "TEST GOALS TEST CLASS ",tclass
        DISPLAY goal_num, goal_descr, g_history ;
        WHILE test_class = tclass OFF
    WAIT
    ENDIF
ENDIF
ELSE
    && iteration not found
    CLEAR
    @ 5,5
    ? "No test goals found for test class ",tclass
    ?? CHR(7)
    WAIT
ENDIF
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE9.PRG
* Author       : Ned Davis
* Date        : 4 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for operator output.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

SET MEMOWIDTH TO 25

choice9 = 0

choicela = 1

**** display menu ****

DO WHILE choice9 # 4

choice9 = 0

SET COLOR TO R/BG,GR+/R,G

CLEAR

@ 3,10 TO 23,70 DOUBLE

@ 18,11 TO 18,69 DOUBLE

SET COLOR TO GR+/BG

@ 5,28 SAY "OPERATOR LISTING OPTIONS"

@ 7,25 SAY "Select your choice by number: "

@ 9,25 SAY "1. List by Operator Number"

@ 10,25 SAY "2. List alphabetically"

@ 11,25 SAY "3. List deleted operators"

@ 12,25 SAY "4. Return to Report Menu"

SET COLOR TO W+/BG

@ 17,26 SAY "Enter your choice here: " GET choice9;

PICTURE "99" RANGE 0,4

READ

SET COLOR TO GR+/BG

IF choice9 # 4

choicela = 1

@ 19,15 SAY "1. Brief output: description & history omitted"

@ 20,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choicela ;

PICTURE "99" RANGE 1,2

READ

SET COLOR TO GR+/BG

ENDIF

**** perform choices ****

DO CASE

CASE choice9 = 1

```

USE opstream INDEX op_num
IF choicela = 1
  IF print_it
    LIST FOR operator .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL FOR operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ELSE
  IF print_it
    LIST op_number, op_name, o_history FOR operator .AND. ;
    deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL op_number, op_name, o_history;
    FOR operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ENDIF
CASE choice9 = 2
USE opstream INDEX op_name
IF choicela = 1
  IF print_it
    LIST FOR operator .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL FOR operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ELSE
  IF print_it
    LIST op_name, op_number, o_history ;
    FOR operator .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL op_name, op_number, o_history;
    FOR operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ENDIF
CASE choice9 = 3
DO DEL_OPS
ENDCASE
CLOSE DATABASES
ENDDO

CLEAR
SET MEMOWIDTH TO 30
RETURN

```

```

*****
*   Program Name   : DEL_OPS.PRG
*   Author        : Ned Davis
*   Date          : 6 Oct 90
*   Revised       :
*   Language      : dBase III+
*   Description    : Allows deleted operators to be listed
*                   to screen or printer in either brief or
*                   descriptive format. Called by R_CASE9.PRG.
*                   WARNING - uses global variable print_it
*                   from REPORTS.PRG and choicela from
*                   R_CASE9.PRG.
*****

**** set memory variables ****
del_ch = 0
iter = 0

**** display menu ****
DO WHILE del_ch # 3
  del_ch = 0
  iter = 0
  CLEAR
  SET COLOR TO R/BG
  @ 3,10 TO 16,70 DOUBLE
  SET COLOR TO GR+/BG
  @ 4,25 SAY "DELETED OPERATOR LISTING OPTIONS"
  @ 6,20 SAY "Select your choice by number: "
  @ 8,20 SAY "1. List all deleted operators"
  @ 9,20 SAY "2. List operators deleted in an iteration"
  @ 10,20 SAY "3. Return to Operator Menu"
  SET COLOR TO W+/BG
  @ 12,20 SAY "Enter your choice here: " GET del_ch;
    PICTURE "99" RANGE 1,3
  READ
  SET COLOR TO GR+/BG
  DO CASE
    CASE del_ch = 1                && list all deleted operators
      USE opstream INDEX op_num
      IF choicela = 1
        IF print_it
          LIST FOR operator .AND. deleted # 0 OFF TO PRINT
        ELSE
          DISPLAY ALL FOR operator .AND. deleted #0 OFF
          WAIT
        ENDIF
      ELSE
        IF print_it
          LIST op_number, op_name, o_history FOR operator;
            .AND. deleted # 0 OFF TO PRINT
        ELSE

```

```

        DISPLAY ALL op_number, op_name, o_history;
        FOR operator .AND. deleted # 0 OFF
        WAIT
    ENDIF
ENDIF
CASE del_ch = 2    && list deleted operators for iter x
    SET COLOR TO W+/BG
    @ 14,20 SAY "Enter the iteration to output: ";
        GET iter PICTURE "999" RANGE 1,99
    READ
    SET COLOR TO GR+/BG
    USE opstream INDEX delet_op
    SEEK iter
    IF FOUND()
        IF choicela = 1
            IF print_it
                LIST WHILE deleted = iter FOR operator;
                    OFF TO PRINT
            ELSE
                DISPLAY WHILE deleted = iter FOR operator OFF
                WAIT
            ENDIF
        ELSE
            IF print_it
                SET PRINT ON
                ? "OPERATORS DELETED FOR ITERATION ",iter
                SET PRINT OFF
                LIST op_number, op_name, o_history;
                    WHILE deleted = iter FOR operator;
                        OFF TO PRINT
            ELSE
                ? "OPERATORS DELETED FOR ITERATION ",iter
                DISPLAY op_number, op_name, o_history;
                    WHILE deleted = iter FOR operator OFF
                WAIT
            ENDIF
        ENDIF
    ELSE
        CLEAR
        @ 5,5
        ? "No deleted operators found for iteration ",iter
        WAIT
    ENDIF
ENDCASE    && return to caller
CLOSE DATABASES
ENDDO
RETURN

```

```

*****
*   Program Name   : R_CASE10.PRG
*   Author        : Ned Davis
*   Date          : 5 Oct 90
*   Revised       :
*   Language      : Dbase III+
*   Description    : Provides menu options for data stream output.
*                   Sends output to screen or printer as req'd.
*                   Called by REPORTS.PRG. !!WARNING- uses global
*                   variable "print_it" from calling program to
*                   det'm where to send output.
*****

**** set environment ****
SET MEMOWIDTH TO 25
choicel0= 0
choicela = 1

**** display menu ****
DO WHILE choicel0 # 4
    choicel0 = 0
    SET COLOR TO R/BG,GR+/R,G
    CLEAR
    @ 3,10 TO 23,70 DOUBLE
    @ 18,11 TO 18,69 DOUBLE
    SET COLOR TO GR+/BG
    @ 5,26 SAY "DATA STREAM LISTING OPTIONS"
    @ 7,25 SAY "Select your choice by number: "
    @ 9,25 SAY "1. List by Data Stream Number"
    @ 10,25 SAY "2. List alphabetically"
    @ 11,25 SAY "3. List deleted Data Streams"
    @ 12,25 SAY "4. Return to Report Menu"
    SET COLOR TO W+/BG
    @ 17,25 SAY "Enter your choice here: " GET choicel0 PICTURE "99";
        RANGE 1,4
    READ
    SET COLOR TO GR+/BG
    IF choicel0 # 4
        choicela = 1
        @ 19,15 SAY "1. Brief output: description & history omitted"
        @ 20,15 SAY "2. Descriptive output: w/ description & history"
        SET COLOR TO W+/BG
        @ 21,15 SAY "Enter your choice here: " GET choicela ;
            PICTURE "99" RANGE 1,2
        READ
        SET COLOR TO GR+/BG
    ENDIF

**** perform choices ****
DO CASE
    CASE choicel0 = 1

```

```

USE opstream INDEX op_num
IF choicela = 1
  IF print_it
    LIST FOR .NOT. operator .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL FOR .NOT. operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ELSE
  IF print_it
    LIST op_number, op_name, o_history FOR .NOT. operator;
    .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL op_number, op_name, o_history;
    FOR .NOT. operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ENDIF
CASE choicel0 = 2
USE opstream INDEX op_name
IF choicela = 1
  IF print_it
    LIST FOR .NOT. operator .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL FOR .NOT. operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ELSE
  IF print_it
    LIST op_name, op_number, o_history ;
    FOR .NOT. operator .AND. deleted = 0 OFF TO PRINT
  ELSE
    DISPLAY ALL op_name, op_number, o_history;
    FOR .NOT. operator .AND. deleted = 0 OFF
    WAIT
  ENDIF
ENDIF
CASE choicel0 = 3
DO DEL_STMS
ENDCASE
CLOSE DATABASES
ENDDO

CLEAR
SET MEMOWIDTH TO 30
RETURN

```

```

*****
* Program Name   : DEL_STMS.PRG
* Author        : Ned Davis
* Date          : 7 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Allows deleted data streams to be listed
*                  to screen or printer in either brief or
*                  descriptive format. Called by R_CASE10.PRG.
*                  WARNING - uses global variable print_it
*                  from REPORTS.PRG and choicela from
*                  R_CASE10.PRG.
*****

```

```

**** set memory variables ****

```

```

del_ch = 0

```

```

iter = 0

```

```

**** display menu ****

```

```

DO WHILE del_ch # 3

```

```

    del_ch = 0

```

```

    iter = 0

```

```

    CLEAR

```

```

    SET COLOR TO R/BG

```

```

    @ 3,10 TO 16,70 DOUBLE

```

```

    SET COLOR TO GR+/BG

```

```

    @ 4,23 SAY "DELETED DATA STREAMS LISTING OPTIONS"

```

```

    @ 6,20 SAY "Select your choice by number: "

```

```

    @ 8,20 SAY "1. List all deleted data streams"

```

```

    @ 9,20 SAY "2. List data streams deleted in an iteration"

```

```

    @ 10,20 SAY "3. Return to Data Stream Menu"

```

```

    SET COLOR TO W+/BG

```

```

    @ 12,20 SAY "Enter your choice here: " GET del_ch;

```

```

        PICTURE "99" RANGE 1,3

```

```

    READ

```

```

    SET COLOR TO GR+/BG

```

```

    DO CASE

```

```

        CASE del_ch = 1          && list all deleted data streams

```

```

            USE opstream INDEX op_num

```

```

            IF choicela = 1

```

```

                IF print_it

```

```

                    LIST FOR .NOT. operator .AND. deleted # 0 OFF TO PRINT

```

```

                ELSE

```

```

                    DISPLAY ALL FOR .NOT. operator .AND. deleted #0 OFF

```

```

                    WAIT

```

```

                ENDIF

```

```

            ELSE

```

```

                IF print_it

```

```

                    LIST op_number, op_name, o_history FOR .NOT. operator;

```

```

                    .AND. deleted # 0 OFF TO PRINT

```

```

                ELSE

```

```

        DISPLAY ALL op_number, op_name, o_history;
        FOR .NOT. operator .AND. deleted # 0 OFF
        WAIT
    ENDIF
ENDIF
CASE del_ch = 2    && list deleted data streams for iter x
    SET COLOR TO W+/BG
    @ 14,20 SAY "Enter the iteration to output: ";
        GET iter PICTURE "999" RANGE 1,99

    READ
    SET COLOR TO GR+/BG
    USE opstream INDEX delet_op
    SEEK iter
    IF FOUND()
        IF choicela = 1
            IF print_it
                LIST WHILE deleted = iter FOR .NOT. operator;
                    OFF TO PRINT
            ELSE
                DISPLAY WHILE deleted = iter FOR .NOT. operator;
                    FOR operator OFF
                WAIT
            ENDIF
        ELSE
            IF print_it
                SET PRINT ON
                ? "OPERATORS DELETED FOR ITERATION ",iter
                SET PRINT OFF
                LIST op_number, op_name, o_history;
                    WHILE deleted = iter FOR .NOT. operator;
                        OFF TO PRINT
            ELSE
                ? "OPERATORS DELETED FOR ITERATION ",iter
                DISPLAY op_number, op_name, o_history;
                    WHILE deleted = iter FOR .NOT. operator OFF
                WAIT
            ENDIF
        ENDIF
    ELSE
        CLEAR
        @ 5,5
        ? "No deleted data streams found for iteration ",iter
        WAIT
    ENDIF
ENDCASE    && return to caller
CLOSE DATABASES
ENDDO
RETURN

```

```
* Program Name : R_CASE11.PRG
* Author       : Ned Davis
* Date        : 5 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for operator output.
*               Outputs operator/data stream added for
*               iteration x.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

SFT MEMOWIDTH TO 25

choicell = 0

choicela = 1

iter = 0

**** display menu ****

DO WHILE choicell # 3

choicell = 0

iter = 0

SET COLOR TO R/BG,GR+/R,G

CLEAR

@ 3,10 TO 23,70 DOUBLE

@ 18,11 TO 18,69 DOUBLE

@ 14,11 TO 14,69 DOUBLE

SET COLOR TO GR+/BG

@ 5,16 SAY "OPERATORS/DATA STREAMS ADDED FOR A GIVEN ITERATION"

@ 7,25 SAY "Select your choice by number: "

@ 9,25 SAY "1. List Operators"

@ 10,25 SAY "2. List Data Streams"

@ 13,25 SAY "3. Return to Report Menu"

SET COLOR TO W+/BG

@ 15,24 SAY "Enter your choice here: " GET choicell PICTURE "99";
RANGE 0,3

READ

@ 17,15 SAY "Enter the iteration to output: " GET iter;
PICTURE "999" RANGE 0,99

SET COLOR TO GR+/BG

IF choicell # 3

choicela = 1

@ 19,15 SAY "1. Brief output: description & history omitted"

@ 20,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choicela ;
PICTURE "99" RANGE 1,2

READ

```

    SET COLOR TO GR+/BG
ENDIF

**** perform choices ****
USE opstream INDEX o_iteradd
SEEK iter
IF FOUND()
    DO CASE
        CASE choicell = 1          && list operators
            IF choicela = 1        && brief description
                IF print_it
                    LIST WHILE o_iteradd = iter FOR operator OFF TO PRINT
                ELSE
                    DISPLAY WHILE o_iteradd = iter FOR operator OFF
                    WAIT
                ENDIF
            ELSE
                && detailed description
                IF print_it
                    SET PRINT ON
                    ? "OPERATORS ADDED IN ITERATION ", iter
                    SET PRINT OFF
                    LIST op_number, op_name, o_history;
                    WHILE o_iteradd = iter FOR operator OFF TO PRINT
                ELSE
                    ? "OPERATORS ADDED IN ITERATION ", iter
                    DISPLAY op_number, op_name, o_history;
                    WHILE o_iteradd = iter FOR operator OFF
                    WAIT
                ENDIF
            ENDIF
        CASE choicell = 2          && list data streams
            IF choicela = 1        && brief description
                IF print_it
                    LIST WHILE o_iteradd = iter;
                    FOR .NOT. operator OFF TO PRINT
                ELSE
                    DISPLAY WHILE o_iteradd = iter FOR .NOT. operator OFF
                    WAIT
                ENDIF
            ELSE
                && detailed description
                IF print_it
                    SET PRINT ON
                    ? "DATA STREAMS ADDED IN ITERATION ", iter
                    SET PRINT OFF
                    LIST op_name, op_number, o_history ;
                    WHILE o_iteradd = iter FOR .NOT. operator OFF TO PRINT
                ELSE
                    ? "DATA STREAMS ADDED IN ITERATION ", iter
                    DISPLAY op_name, op_number, o_history;
                    WHILE o_iteradd = iter FOR .NOT. operator OFF
                    WAIT
            ENDIF
    ENDIF

```

```
* Program Name : R_CASE12.PRG
* Author      : Ned Davis
* Date       : 5 Oct 90
* Revised    :
* Language   : Dbase III+
* Description : Provides menu options for operator output.
*             Outputs operator/data stream changed for
*             iteration x.
*             Sends output to screen or printer as req'd.
*             Called by REPORTS.PRG. !!WARNING- uses global
*             variable "print_it" from calling program to
*             det'm where to send output.
```

**** set environment ****

```
SET MEMOWIDTH TO 25
choice12 = 0
choicela = 1
ITER = 0
```

**** display menu ****

```
DO WHILE choice12 # 3
  choice12 = 0
  iter = 0
  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 23,70 DOUBLE
  @ 18,11 TO 18,69 DOUBLE
  @ 14,11 TO 14,69 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,14 SAY "OPERATORS/DATA STREAMS CHANGED FOR A GIVEN ITERATION"
  @ 7,25 SAY "Select your choice by number: "
  @ 9,25 SAY "1. List Operators"
  @ 10,25 SAY "2. List Data Streams"
  @ 13,25 SAY "3. Return to Report Menu"
  SET COLOR TO W+/BG
  @ 15,25 SAY "Enter your choice here: " GET choice12 PICTURE "99";
    RANGE 0,3
  READ
  @ 17,15 SAY "Enter the iteration to output: " GET iter;
    PICTURE "999" RANGE 0,99
  SET COLOR TO GR+/BG
  IF choice12 # 3
    choicela = 1
    @ 19,15 SAY "1. Brief output: description & history omitted"
    @ 20,15 SAY "2. Descriptive output: w/ description & history"
    SET COLOR TO W+/BG
    @ 21,15 SAY "Enter your choice here: " GET choicela ;
      PICTURE "99" RANGE 1,2
  READ
```

```

    SET COLOR TO GR+/BG
ENDIF

**** perform choices ****
USE opstream INDEX o_update
SEEK iter
IF FOUND()
    DO CASE
        CASE choicel2 = 1          && list operators
            IF choicela = 1        && brief description
                IF print_it
                    LIST WHILE updated = iter FOR operator OFF TO PRINT
                ELSE
                    DISPLAY WHILE updated = iter FOR operator OFF
                    WAIT
                ENDIF
            ELSE
                && detailed description
                IF print_it
                    SET PRINT ON
                    ? "OPERATORS CHANGED IN ITERATION ",iter
                    SET PRINT OFF
                    LIST op_number, op_name, o_history;
                    WHILE updated = iter FOR operator OFF TO PRINT
                ELSE
                    ? "OPERATORS CHANGED IN ITERATION ",iter
                    DISPLAY op_number, op_name, o_history;
                    WHILE updated = iter FOR operator OFF
                    WAIT
                ENDIF
            ENDIF
        CASE choicel2 = 2          && list data streams
            IF choicela = 1        && brief description
                IF print_it
                    LIST WHILE updated = iter;
                    FOR .NOT. operator OFF TO PRINT
                ELSE
                    DISPLAY WHILE updated = iter FOR .NOT. operator OFF
                    WAIT
                ENDIF
            ELSE
                && detailed description
                IF print_it
                    SET PRINT ON
                    ? "DATA STREAMS UPDATED IN ITERATION ", iter
                    SET PRINT OFF
                    LIST op_name, op_number, o_history ;
                    WHILE updated = iter FOR .NOT. operator OFF TO PRINT
                ELSE
                    ? "DATA STREAMS CHANGED IN ITERATION ", iter
                    DISPLAY op_name, op_number, o_history;
                    WHILE updated = iter FOR .NOT. operator OFF
                    WAIT
            ENDIF
    ENDIF

```

```

        ENDIF
    ENDIF
ENDCASE
ELSE
    IF iter # 0
        CLEAR
        @ 5,5
        IF choice12 = 1
            ? "No operators found for iteration ",iter
        ELSE
            ? "No data streams found for iteration ", iter
        ENDIF
        ??CHR(7)
        WAIT
    ENDIF
ENDIF
CLOSE DATABASES
ENDDO

CLEAR
SET MEMOWIDTH TO 30
RETURN

```

```
* Program Name : R_CASE13.PRG
* Author      : Ned Davis
* Date       : 5 Oct 90
* Revised    :
* Language   : Dbase III+
* Description : Provides menu options for operator output.
*              Outputs operator/data stream deleted for
*              iteration x.
*              Sends output to screen or printer as req'd.
*              Called by REPORTS.PRG. !!WARNING- uses global
*              variable "print_it" from calling program to
*              det'm where to send output.
```

**** set environment ****

SET MEMOWIDTH TO 25

choice13= 0

choicela = 1

iter = 0

**** display menu ****

DO WHILE choice13 # 3

choice13 = 0

iter = 0

SET COLOR TO R/BG,GR+/R,G

CLEAR

@ 3,10 TO 23,70 DOUBLE

@ 18,11 TO 18,69 DOUBLE

@ 14,11 TO 14,69 DOUBLE

SET COLOR TO GR+/BG

@ 5,14 SAY "OPERATORS/DATA STREAMS DELETED FOR A GIVEN ITERATION"

@ 7,25 SAY "Select your choice by number: "

@ 9,25 SAY "1. List Operators"

@ 10,25 SAY "2. List Data Streams"

@ 13,25 SAY "3. Return to Report Menu"

SET COLOR TO W+/BG

@ 15,25 SAY "Enter your choice here: " GET choice13 PICTURE "99";
RANGE 0,3

READ

@ 17,15 SAY "Enter the iteration to output: " GET iter;
PICTURE "999" RANGE 0,99

SET COLOR TO GR+/BG

IF choice13 # 3

choicela = 1

@ 19,15 SAY "1. Brief output: description & history omitted"

@ 20,15 SAY "2. Descriptive output: w/ description & history"

SET COLOR TO W+/BG

@ 21,15 SAY "Enter your choice here: " GET choicela ;
PICTURE "99" RANGE 1,2

READ

```

    SET COLOR TO GR+/BG
ENDIF

**** perform choices ****
USE opstream INDEX delet_op
SEEK iter
IF FOUND()
    DO CASE
        CASE choicel3 = 1          && list operators
            IF choicela = 1        && brief description
                IF print_it
                    LIST WHILE deleted = iter FOR operator OFF TO PRINT
                ELSE
                    DISPLAY WHILE deleted = iter FOR operator OFF
                    WAIT
                ENDIF
            ELSE
                && detailed description
                IF print_it
                    SET PRINT ON
                    ? "OPERATORS DELETED IN ITERATION ",iter
                    SET PRINT OFF
                    LIST op_number, op_name, o_history;
                    WHILE deleted = iter FOR operator OFF TO PRINT
                ELSE
                    ? "OPERATORS DELETED IN ITERATION ",iter
                    DISPLAY op_number, op_name, o_history;
                    WHILE deleted = iter FOR operator OFF
                    WAIT
                ENDIF
            ENDIF
        CASE choicel3 = 2          && list data streams
            IF choicela = 1        && brief description
                IF print_it
                    LIST WHILE deleted = iter;
                    FOR .NOT. operator OFF TO PRINT
                ELSE
                    DISPLAY WHILE deleted = iter FOR .NOT. operator OFF
                    WAIT
                ENDIF
            ELSE
                && detailed description
                IF print_it
                    SET PRINT ON
                    ? "DATA STREAMS DELETED IN ITERATION ", iter
                    SET PRINT OFF
                    LIST op_name, op_number, o_history ;
                    WHILE deleted = iter FOR .NOT. operator OFF TO PRINT
                ELSE
                    ? "DATA STREAMS DELETED IN ITERATION ", iter
                    DISPLAY op_name, op_number, o_history;
                    WHILE deleted = iter FOR .NOT. operator OFF
                    WAIT

```

```

        ENDIF
    ENDIF
ENDCASE
ELSE
    IF iter # 0
        CLEAR
        @ 5,5
        IF choice13 = 1
            ? "No operators found for iteration ",iter
        ELSE
            ? "No data streams found for iteration ",iter
        ENDIF
        ??CHR(7)
        WAIT
    ENDIF
ENDIF
CLOSE DATABASES
ENDDO

CLEAR
SET MEMOWIDTH TO 30
RETURN

```

```
* Program Name : R_CASE14.PRG
* Author       : Ned Davis
* Date        : 9 Oct 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for op/data stream output.
*               Outputs op/data streams that map to a particular
*               test goal - either all or for an iteration.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

```
choicel4 = 0
iter_no = 0
choicela = 0
tgoal = 0
SET MEMOWIDTH TO 20
```

**** display menu ****

```
DO WHILE choicel4 # 3
  choicel4 = 0
  iter_no = 0
  choicela = 0
  tgoal = 0

  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 23,70 DOUBLE
  @ 14,11 TO 14,69 DOUBLE
  @ 17,11 TO 17,69 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,15 SAY "OPS/DATA STREAMS THAT MAP TO A GIVEN TEST GOAL"
  @ 7,20 SAY "Select your choice by number: "
  @ 9,20 SAY "1. List Ops/Data Streams for all iterations"
  @ 10,20 SAY "2. List Ops/Data Streams for a given iteration"
  @ 11,20 SAY "3. Return to Report Menu"
  SET COLOR TO W+/BG
  @ 13,20 SAY "Enter your choice here: " GET choicel4;
    PICTURE "99" RANGE 0,3
  READ
  SET COLOR TO GR+/BG
  IF choicel4 = 2
    @ 15,20 SAY "Enter the iteration to output or 0 to escape"
    SET COLOR TO W+/BG
    @ 16,20 SAY "Enter your choice here: " GET iter_no PICTURE "999";
      RANGE 0,99
  READ
```

```

    SET COLOR TO GR+/BG
ENDIF
IF iter_no # 0 .OR. choicel4 # 3
    choicela = 1
    @ 18,15 SAY "1. Brief output: description & history omitted"
    @ 19,15 SAY "2. Descriptive output: w/ description & history"
    SET COLOR TO W+/BG
    @ 21,15 SAY "Enter your choice here: " GET choicela ;
        PICTURE "99" RANGE 1,2

    READ
    @ 22,15 SAY "Enter the test goal for op/data stream listing: ";
        GET tgoal PICTURE "99999" RANGE 1,9999

    READ
    SET COLOR TO GR+/BG
ENDIF

**** perform choices ****
CLEAR
DO CASE
    CASE choicel4 = 1
        DO TRACE1TG
    CASE choicel4 = 2
        DO TRACE2TG
ENDCASE
CLOSE DATABASES
ENDDO
SET MEMOWIDTH TO 30
CLEAR
RETURN

```

```

*****
*   Program Name   : TRACE1TG.PRG
*   Author        : Ned Davis
*   Date          : 9 Oct 90
*   Revised       :
*   Language      : dBase III+
*   Description    : Prints to screen or printer a listing of
*                   op/data strms that link to a particular
*                   test goal. Called from R_CASE14.PRG.
*                   WARNING!! - uses vars from R_CASE14.PRG.
*****

```

**** set work areas ****

```

SELECT A
USE ro_link INDEX gnumlink

```

```

SELECT B
USE opstream INDEX op_num

```

**** set up relationships ****

```

SELECT A
SET RELATION TO op_number INTO opstream

```

**** output report ****

```

GO TOP
SEEK tgoal
IF FOUND()
    IF choicela = 1                && brief format
        IF print_it
            SET PRINT ON
            ? "OP/DATA STREAMS THAT MAP TO TEST GOAL ",tgoal
            SET PRINT OFF
            DO WHILE goal_num = tgoal
                SELECT B
                LIST FOR op_number = A->op_number OFF TO PRINT
                SELECT A
                SKIP
            ENDDO
        ELSE
            ? "OP/DATA STREAMS THAT MAP TO TEST GOAL",tgoal
            DO WHILE goal_num = tgoal
                SELECT B
                DISPLAY FOR op_number = A->op_number OFF
                SELECT A
                SKIP
            ENDDO
            WAIT
        ENDIF
    ELSE                            && detailed format
        IF print_it
            SET PRINT ON

```

```

? "OP/DATA STREAMS THAT MAP TO TEST GOAL ",tgoal
SET PRINT OFF
DO WHILE goal_num = tgoal
    SELECT B
    LIST op_number, op_name, o_history;
    FOR op_number = A->op_number OFF TO PRINT
    SELECT A
    SKIP
ENDDO
ELSE
? "OP/DATA STREAMS THAT MAP TO TEST GOAL ",tgoal
DO WHILE goal_num = tgoal
    SELECT B
    DISPLAY op_number, op_name, o_history;
    FOR op_number = A->op_number OFF
    SELECT A
    SKIP
ENDDO
WAIT
ENDIF
ENDIF
ELSE
&& tgoal not found
CLEAR
@ 5,5
? "No test goal found to match ",tgoal
WAIT
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```

*****
* Program Name   : TRACE2TG.PRG
* Author        : Ned Davis
* Date          : 9 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Prints to screen or printer a listing of
*                  op/data strms that link to a particular
*                  test goal, where the links were added in a
*                  given iteration. Called from R_CASE14.PRG.
*                  WARNING!! - uses vars from R_CASE14.PRG.
*****

```

**** set work areas ****

```

SELECT A
USE ro_link INDEX gnumlink

```

```

SELECT B
USE opstream INDEX op_num

```

**** set up relationships ****

```

SELECT A
SET RELATION TO op_number INTO opstream

```

**** output report ****

```

GO TOP
SEEK tgoal
IF FOUND()
  IF choicela = 1          && brief format
    IF print_it
      SET PRINT ON
      ? "OP/DATA STREAMS THAT MAP TO TEST GOAL ",tgoal
      ?? " FOR ITERATION ", iter_no
      SET PRINT OFF
      DO WHILE goal_num = tgoal
        IF l_iteradd = iter_no
          SELECT B
          LIST FOR op_number = A->op_number OFF TO PRINT
          SELECT A
        ENDIF
        SKIP
      ENDDO
    ELSE
      ? "OP/DATA STREAMS THAT MAP TO TEST GOAL",tgoal
      ?? " FOR ITERATION ",iter_no
      DO WHILE goal_num = tgoal
        IF l_iteradd = iter_no
          SELECT B
          DISPLAY FOR op_number = A->op_number OFF
          SELECT A
        ENDIF
      ENDDO
    ENDIF
  ENDIF

```

```

        SKIP
    ENDDO
    WAIT
ENDIF
ELSE
        && detailed format
    IF print_it
        SET PRINT ON
        ? "OP/DATA STREAMS THAT MAP TO TEST GOAL ",tgoal
        ?? " FOR ITERATION ",iter_no
        SET PRINT OFF
        DO WHILE goal_num = tgoal
            IF l_iteradd = iter_no
                SELECT B
                LIST op_number, op_name, o_history;
                FOR op_number = A->op_number OFF TO PRINT
                SELECT A
            ENDIF
            SKIP
        ENDDO
    ELSE
        ? " OP/DATA STREAMS THAT MAP TO TEST GOAL ",tgoal
        ?? " FOR ITERATION ", iter_no
        DO WHILE goal_num = tgoal
            IF l_iteradd = iter_no
                SELECT B
                DISPLAY op_number, op_name, o_history;
                FOR op_number = A->op_number OFF
                SELECT A
            ENDIF
            SKIP
        ENDDO
        WAIT
    ENDIF
ENDIF
ELSE
        && opnum not found
    CLEAR
    @ 5,5
    ? "No test goal found to match ",tgoal
    ? " for iteration ",iter_no
    WAIT
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program Name : R_CASE15.PRG
* Author      : Ned Davis
* Date       : 9 Oct. 90
* Revised    :
* Language   : Dbase III+
* Description : Provides menu options for link output.
*             Sends output to screen or printer as req'd.
*             Called by REPORTS.PRG. !!WARNING- uses global
*             variable "print_it" from calling program to
*             det'm where to send output.
```

**** set environment ****

```
choicel = 0
choicela = 1
gnum = 0
onum = 0
ppart = "
```

**** display menu ****

```
DO WHILE choicel # 6
  choicel = 0
  gnum = 0
  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 23,70 DOUBLE
  @ 18,11 TO 18,69 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,27 SAY "GLOBAL LINK LISTING OPTIONS"
  @ 7,20 SAY "Select your choice by number: "
  @ 9,20 SAY "1. List all Links"
  @ 10,20 SAY "2. List Links for a Test Goal Number"
  @ 11,20 SAY "3. List Links for an Op/Data Strm Number"
  @ 12,20 SAY "4. List Links for a PSDL Part"
  @ 13,20 SAY "5. List all deleted Links"
  @ 14,20 SAY "6. Return to Report Menu"
  SET COLOR TO W+/BG
  @ 17,20 SAY "Enter your choice here: " GET choicel PICTURE "99";
  RANGE 0,6
  READ
  SET COLOR TO GR+/BG
```

**** perform choices ****

```
DO CASE
  CASE choicel = 1      && list all links
    USE ro_link INDEX gnumlink
    IF print_it
      LIST FOR deleted = 0 OFF TO PRINT
    ELSE
      DISPLAY ALL FOR deleted = 0 OFF
```

```

        WAIT
    ENDIF
CASE choice1 = 2      && list links for a goal no.
    USE ro_link INDEX gnumlink
    @ 19,25 SAY "Enter goal number to seek: " GET gnum;
    PICTURE "99999" RANGE 1,9999
    READ
    SEEK gnum
    IF FOUND()
        CLEAR
        IF print_it
            LIST WHILE goal_num = gnum FOR deleted = 0 OFF TO PRINT
        ELSE
            DISPLAY WHILE goal_num = gnum FOR deleted = 0 OFF
        WAIT
    ENDIF
ELSE
    CLEAR
    @ 5,5
    ? "No links found for goal number ",gnum
    WAIT
ENDIF
CASE choice1 = 3      && list links for an op no.
    DO LINK4OPS
CASE choice1 = 4      && list links for PSDL part
    DO LINK4PRT
CASE choice1 = 5      && list deleted links
    USE ro_link INDEX delet_lk
    IF print_it
        LIST FOR deleted # 0 OFF TO PRINT
    ELSE
        DISPLAY ALL FOR deleted # 0 OFF
    WAIT
ENDIF
ENDCASE
CLOSE DATABASES
ENDDO
CLEAR
RETURN

```

```

*****
* Program Name : LINK4OPS.PRG
* Author      : Ned Davis
* Date       : 9 Oct 90
* Revised    :
* Language   : Dbase III+
* Description : Lists links for an operator number.
*             Called by REPORTS.PRG. !!WARNING- uses global
*             variable "print_it" from calling program to
*             det'm where to send output.
*****

USE ro_link INDEX onumlink
@ 19,16 SAY "Enter operator/data stream number to seek: " GET onum;
    PICTURE "99999" RANGE 1,9999
READ
SEEK onum
IF FOUND()
    CLEAR
    IF print_it
        LIST WHILE op_number = onum FOR deleted = 0 OFF TO PRINT
    ELSE
        DISPLAY WHILE op_number = onum FOR deleted = 0 OFF
        WAIT
    ENDIF
ELSE
    CLEAR
    @ 5,5
    ? "No links found for operator/data stream number ",onum
    WAIT
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```

*****
* Program Name : LINK4PRT.PRG
* Author      : Ned Davis
* Date       : 10 Oct 90
* Revised    :
* Language   : Dbase III+
* Description : Lists links for a PSDL part.
*             Called by R_CASE15.PRG. !!WARNING- uses global
*             variable "print_it" from calling program to
*             det'm where to send output.
*****

```

```

USE ro_link INDEX partlink
@ 19,12 SAY "Enter PSDL part: "GET ppart;
    PICTURE "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
READ
SEEK ppart
IF FOUND()
    CLEAR
    IF print_it
        LIST WHILE psdl_part = ppart FOR deleted = 0 OFF TO PRINT
    ELSE
        DISPLAY WHILE psdl_part = ppart FOR deleted = 0 OFF
        WAIT
    ENDIF
ELSE
    CLEAR
    @ 5,5
    ? "No links found for PSDL part ",ppart
    WAIT
ENDIF
CLOSE DATABASES
CLEAR
RETURN

```

```
* Program      : R_CASE16.PRG
* Author       : Ned Davis
* Date        : 11 Oct. 90
* Revised     :
* Language    : Dbase III+
* Description  : Provides menu options for link output for
*               a given iteration.
*               Sends output to screen or printer as req'd.
*               Called by REPORTS.PRG. !!WARNING- uses global
*               variable "print_it" from calling program to
*               det'm where to send output.
```

**** set environment ****

```
listing = 0
choice2 = 1
iter_no = 0
```

**** display menu ****

DO WHILE listing # 3

```
    listing = 0
    choice2 = 0
    iter_no = 0
```

SET COLOR TO R/BG,GR+/R,G

CLEAR

@ 3,10 TO 18,70 DOUBLE

@ 11,11 TO 11,69 DOUBLE

@ 14,11 TO 14,69 DOUBLE

SET COLOR TO GR+/BG

@ 5,20 SAY "LINK OUTPUT OPTIONS FOR A GIVEN ITERATION"

@ 7,20 SAY "1. List Links added for iteration"

@ 8,20 SAY "2. List Links deleted for iteration"

@ 9,20 SAY "3. Return to Reports Menu"

SET COLOR TO W+/BG

@ 10,20 SAY "Enter your choice here: " GET listing PICTURE "99";
RANGE 1,3

READ

SET COLOR TO GR+/BG

IF listing # 3

@ 12,20 SAY "Enter the iteration to output"

SET COLOR TO W+/BG

@ 13,20 SAY "Enter your choice here: " GET iter_no PICTURE "999";
RANGE 1,99

READ

SET COLOR TO GR+/BG

@ 15,20 SAY "1. List by goal number"

@ 16,20 SAY "2. List by operator/ data stream number"

SET COLOR TO W+/BG

```

@ 17,20 SAY "Enter your choice here: " GET choice2 ;
      PICTURE "99" RANGE 1,2
READ
SET COLOR TO GR+/BG
ENDIF

**** perform choices ****
DO CASE
CASE choice2 = 1
  USE ro_link INDEX gnumlink
  IF listing = 1    && links added
    IF print_it
      LIST FOR l_iteradd = iter_no OFF TO PRINT
    ELSE
      DISPLAY FOR l_iteradd = iter_no OFF
      WAIT
    ENDIF
  ELSE
    && links deleted
    IF print_it
      LIST FOR deleted = iter_no OFF TO PRINT
    ELSE
      DISPLAY FOR deleted = iter_no OFF
      WAIT
    ENDIF
  ENDIF
CASE choice2 = 2
  USE ro_link INDEX onumlink
  IF listing = 1    && links added
    IF print_it
      LIST OP_NUMBER, GOAL_NUM, PSDL_PART, DELETED, L_ITERADD;
      FOR l_iteradd = iter_no OFF TO PRINT
    ELSE
      DISPLAY OP_NUMBER, GOAL_NUM, PSDL_PART, DELETED, i
L_ITERADD;
      FOR l_iteradd = iter_no OFF
      WAIT
    ENDIF
  ELSE
    && links deleted
    IF print_it
      LIST OP_NUMBER, GOAL_NUM, PSDL_PART, DELETED, i
L_ITERADD;
      FOR deleted = iter_no OFF TO PRINT
    ELSE
      DISPLAY OP_NUMBER, GOAL_NUM, PSDL_PART, DELETED, i
L_ITERADD;
      FOR deleted = iter_no OFF
      WAIT
    ENDIF
  ENDIF
ENDCASE
CLOSE DATABASES

```

ENDDO
CLEAR
RETURN

```

*****
* Program Name : UP_GOALS.PRG
* Author      : Ned Davis
* Date       : 17 Aug 90
* Revised    : 31 Aug 90 ~ Mapping option added.
* Language   : Dbase III+
* Description : Provides the menu screen for allowing update
*              of the test goals database file and allows the
*              user to move to other system actions in the
*              TGTS for requirements-based testing in the
*              CAPS.
*****

```

****set environment****

choice = 0

****display test goal update menu****

```

DO WHILE choice # 6
  choice = 0
  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 20,70 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,30 SAY "TEST GOAL UPDATE MENU"
  @ 7,20 SAY "Select your choice by number: "
  @ 9,20 SAY "1. Add a new test goal"
  @ 10,20 SAY "2. Change an existing test goal"
  @ 11,20 SAY "3. Delete an existing test goal"
  @ 12,20 SAY "4. Link test goal to PSDL operator/data stream"
  @ 13,20 SAY "5. Unlink test goal to PSDL operator/data stream"
  @ 14,20 SAY "6. Return to Main Menu"
  SET COLOR TO W+/BG
  @ 17,15 SAY "Select your option: " GET choice PICTURE "99" ;
  RANGE 0,6
  READ

```

****perform user's request****

```

DO CASE
  CASE choice = 1
    DO ADD_TG
  CASE choice = 2
    DO CHANGETG
  CASE choice = 3
    DO DEL_TG
  CASE choice = 4
    DO ADD_LINK
  CASE choice = 5
    DO DEL_LINK
ENDCASE

```

ENDDO

CLEAR
choice = 0
RETURN TO MASTER

```

*****
*   Program Name   :   ADD_TG.PRG
*   Author        :   Ned Davis
*   Date          :   17-27 Aug 90
*   Revised       :
*   Language      :   Dbase III+
*   Description    :   Allows the addition of one or more new test
*                       goals to the test goal database.
*****

**** open database ****
USE goals

**** find largest current goal num ****
GO BOTTOM
Mgoal_num = goal_num      && assumes largest num is last

**** select entry screen ****
SET FORMAT TO addtgscn

**** set index files ****
SET INDEX TO goal_num, test_pri, aggregat, updated, iter_add,;
             testclas, delet_tg

**** start loop for adding goals ****
Adding = " "      && memvar for accept, cancel, exit entry scrn
DO WHILE Adding # "X"

    ****initialize memory variables****
    MTest_pri = 0
    MAggregate = 0
    MUpdated = 0
    MIter_add = 0
    MTest_clas = 0
    MDeleted = 0
    Adding = " "
    MGoal_num = MGoal_num + 1

    APPEND BLANK
    *--read in values using addtgscn.fmt
    READ

    IF adding # "C"
        IF adding # "X" .OR. (adding = "X" .AND. MIter_add # 0)

            *--store memory vars on new record
            REPLACE goal_num WITH MGoal_num, ;
                    test_pri WITH MTest_pri, ;
                    aggregate WITH MAggregate, ;
                    updated WITH MUpdated
            REPLACE iter_added WITH MIter_add, ;

```

```

        test_class WITH MTest_clas, ;
        deleted WITH MDeleted
    ELSE
        DELETE
        PACK
    ENDIF
ELSE
    DELETE
    PACK
    MGoal_num = MGoal_num - 1  && decrement count; canc'd rec.
ENDIF
ENDDO
CLOSE DATABASES
CLOSE FORMAT
RETURN

```

```
* Program Name:  ADDTGSCN.FMT
* Author       :  Ned Davis
* Date        :  24 Aug 90
* Revised     :  31 Aug 90
* Language    :  dBase III+
* Description  :  Format file for entering new test goals to
*                  the goals.dbf database file.  Called by
*                  ADD_TG.PRG.  Does not allow update or deletion
*                  to be flagged.
```

```
@ 1, 25 SAY "TEST GOALS DATABASE DATA ENTRY"
@ 3, 0  SAY "TEST GOAL NUMBER"
*--auto insert next goal number
@ 3, 20 SAY MGOAL_NUM
@ 6, 0  SAY "GOAL DESCRIPTION"
@ 6, 18 GET GOALS->GOAL_DESCR
@ 6, 23 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 8, 0  SAY "TEST PRIORITY"
@ 8, 15 GET MTEST_PRI  PICTURE "99"      RANGE 0,4
@ 8, 28 SAY "AGGREGATE"
@ 8, 39 GET MAGGREGATE PICTURE "999"     RANGE 0,99
@ 8, 47 SAY "TEST CLASS"
@ 8, 59 GET MTEST_CLAS PICTURE "99"      RANGE 1,4
@ 10, 0  SAY "ITERATION ADDED"
@ 10, 17 GET MITER_ADD  PICTURE "999"    RANGE 1,99
*@ 12, 0  SAY "ITERATION DELETED"
*@ 12, 19 GET MDELETED  PICTURE "999"    RANGE 0,99
@ 15, 0  SAY "HISTORY"
@ 15, 9  GET GOALS->G_HISTORY
@ 15, 17 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 19, 2  SAY "Press "+CHR(24)+" to make corrections,"
@ 20, 2  SAY "Return to continue, C to Cancel,"
@ 21, 2  SAY "or X to exit";
      GET Adding PICTURE "!!"
@ 0, 0  TO 2, 79  DOUBLE
@ 18, 0  TO 22, 79 DOUBLE
```

```
* Program Name   : CHANGTG.PRG
* Author        : Ned Davis
* Date          : 1 Dec 90
* Revised       :
* Language      : dBase III+
* Description    : Allows user to update 1 record at a time
                  of the GOALS.DBF database. All fields may
                  be modified except goal_num field.
                  A deleted goal may be undeleted and its
                  related link records may be block undeleted
                  or selectively undeleted. Goals are not
                  deleted by this program. Called by
                  UP_GOALS.PRG.
```

**** initialize memory variables ****

```
Adding = "Y"      && loop flag
Change_it = "Y"   && choice variable
Mgoal_num = 0     && dummy var to get goal number
choice_a = 1
```

**** start loop for changing goals ****

```
DO WHILE Adding # "N"
```

```
  CLOSE DATABASES
```

**** select database ****

```
USE goals INDEX i
   goal_num,test_pri,aggregat,updated,iter_add,testclas,delet_tg
```

**** display input screen ****

```
SET FORMAT TO tgchg
READ
```

**** det'm whether to change record ****

```
IF Adding = "N"      && decided not to change record
  CLOSE FORMAT
  LOOP
ENDIF
```

**** prep screen ****

```
CLOSE FORMAT
CLEAR
```

**** look for record ****

```
SEEK Mgoal_num
IF FOUND()
  *--show it to the user for confirmation
  DISPLAY goal_num, goal_descr,g_history OFF

  *--get confirmation
```

```

Change_it = "Y"      && set default
@ 22,0 SAY "Do you want to change this record? (Y or N)"
@ 22,45 GET Change_it PICTURE "Y"
READ

*--act on confirmation
IF Change_it = "N"      && don't change this record
  LOOP
ELSE      && change record
  CLEAR
  IF deleted = 0      && record is in use
    SET FORMAT TO chgtgscn
    READ
    WAIT "Goal updated, press any key to continue"
    CLOSE FORMAT
  ELSE      && record is deleted
    choice_a = 2      && set default choice
    @ 0,0 SAY "Goal is currently deleted"
    @ 3,0 SAY "Options:"
    @ 5,5 SAY "1. Undelete and make appropriate changes"
    @ 6,5 SAY "2. Disregard goal change"
    @ 8,5 SAY "Select your choice: "
    @ 8,25 GET choice_a PICTURE "99" RANGE 1,2
    READ

    **** act upon choice ****
    DO CASE
      CASE choice_a = 1      && undelete
        DO CHGTGHLF
      CASE choice_a = 2      && disregard
        LOOP
    ENDCASE
  ENDIF
ENDIF
ELSE
  *--if not found, warn user
  @ 22,0
  ? "Can't find test goal: ",Mgoal_num
  ?? CHR(7)
  WAIT
ENDIF
Mgoal_num = 0
ENDDO
CLOSE DATABASES
RETURN

```

```
* Program Name   : TGCHG.PMT
* Author        : Ned Davis
* Date          : 7 Sep 90
* Revised       :
* Language      : dBase III+
* Description    : Screen for inputting test goal number to
*                  change or exit w/o change. Starts modification
*                  of GOALS.DBF data base file one record at a
*                  time. May also modify RO_LINK.DBF if a
*                  goal previously deleted is undeleted.
*                  Called by CHANGETG.PRG.
```

```
@ 5, 5 SAY "Enter goal number ( or 0 ) of goal to change:"
@ 5,52 GET Mgoal_num PICTURE "999999"
@ 7, 5 SAY "Enter N to exit w/o change, Y to continue"
@ 7,58 GET Adding PICTURE "Y"
```

```

*****
* Program Name:  CHGTGSCN.FMT
* Author       :  Ned Davis
* Date        :  7 Sep 90
* Revised     :
* Language    :  dBase III+
* Description :  Format file for changing test goals in
*                the goals.dbf database file. Called by
*                CHANGETG.PRG. Does not allow deletion
*                or undeletion.
*****

```

```

@ 1, 25 SAY "TEST GOALS DATABASE DATA CHANGE"
@ 3, 0  SAY "TEST GOAL NUMBER"
@ 3, 20 SAY GOAL_NUM
@ 6, 0  SAY "GOAL DESCRIPTION"
@ 6, 18 GET GOAL_DESCR
@ 8, 0  SAY "TEST PRIORITY"
@ 8, 15 GET TEST_PRI  PICTURE "99"    RANGE 0,4
@ 8, 28 SAY "AGGREGATE"
@ 8, 39 GET AGGREGATE PICTURE "999"   RANGE 0,99
@ 8, 47 SAY "TEST CLASS"
@ 8, 59 GET TEST_CLASS PICTURE "99"   RANGE 0,4
@ 10, 0 SAY "ITERATION ADDED"
@ 10, 17 GET ITER_ADDED PICTURE "999" RANGE 1,99
@ 12, 0 SAY "ITERATION UPDATED"
@ 12, 19 GET UPDATED PICTURE "999"   RANGE 0,99
@ 12, 32 SAY "ITERATION DELETED"
@ 12, 51 SAY DELETED
@ 15, 0 SAY "HISTORY"
@ 15, 9 GET G_HISTORY
@ 15, 17 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 19, 2 SAY "Press "+CHR(24)+" to make corrections,"
*@ 20, 2 SAY "Return to continue, C to Cancel,"
*@ 21, 2 SAY "or X to exit";
* GET Adding PICTURE "!"
@ 0, 0 TO 2, 79 DOUBLE
@ 18, 0 TO 22, 79 DOUBLE

```

```

*****
*   Program Name   :  CHGTGHL.PRG
*   Author        :  Ned Davis
*   Date          :  1 Dec 90
*   Revised       :
*   Language      :  dBase III+
*   Description    :  Allows user to update records and links
*                   :  of the GOALS.DBF database. All fields may
*                   :  be modified except goal_num fields.
*                   :  A deleted goal may be undeleted and its
*                   :  related link records may be block undeleted
*                   :  or selectively undeleted. Goals are not
*                   :  deleted by this program. Called by
*                   :  CHANGETG.PRG. Uses global vars.
*****

**** initialize memory variables ****
Miterchg = 0
ch_link = 1

**** act upon choice ****
SET FORMAT TO chtgscn1
READ
REPLACE deleted WITH 0
CLEAR
@ 5,5
WAIT "Goal updated, press any key to continue"
CLOSE FORMAT

*--undelete link records. Change database.
USE ro_link INDEX gnumlink
SEEK Mgoal_num
IF FOUND()      && link record(s) found
  CLEAR
  ch_link = 1  && set default choice
  @ 5,30 SAY "UNDELETE LINK OPTIONS"
  @ 7,15 SAY "Options:"
  @ 9,15 SAY "1. Batch undelete all links to PSDL part"
  @ 10,15 SAY "2. Sequence through links for selective undelete"
  @ 12,15 SAY "Enter option 1 or 2:"
  @ 12,37 GET ch_link PICTURE "99" RANGE 1,2
  READ

DO CASE
  CASE ch_link = 1
    DO WHILE goal_num = Mgoal_num
      REPLACE deleted WITH 0
      REPLACE l_iteradd WITH Miterchg
      SKIP
    ENDDO
  CASE ch_link = 2

```

```

DO WHILE goal_num = Mgoal_num
  CLEAR
  DISPLAY OFF
  @ 5,5 SAY "Enter Y to undelete, N to skip"
  @ 5,37 GET Adding PICTURE "Y"
  READ
  IF Adding = "Y"
    REPLACE deleted WITH 0
    REPLACE l_iteradd WITH Miterchg
    SKIP
  ELSE
    SKIP
  ENDIF
ENDDO
ENDCASE

**** update index file ****
SET INDEX TO delet_lk
REINDEX
CLOSE INDEX
CLEAR
@ 5,5
WAIT "Links updated, press any key to continue"

ELSE                                && no link records found
  CLEAR
  @ 5,5 SAY "No link record found for this test goal."
  WAIT
ENDIF
CLOSE DATABASES
RETURN

```

```

*****
*   Program Name:  CHTGSCN1.FMT
*   Author       :  Ned Davis
*   Date        :  7 Sep 90
*   Revised     :
*   Language    :  dBase III+
*   Description  :  Format file for changing test goals in
*                   the goals.dbf database file. Called by
*                   CHGETGHLP.PRG. Used explicitly with deleted
*                   goal records that are to be undeleted. Aids
*                   in ensuring that info that must change with
*                   an undelete gets changed.
*****

```

```

@ 1, 20 SAY "TEST GOALS DATABASE DATA CHANGE FOR UNDELETE"
@ 3, 0  SAY "TEST GOAL NUMBER"
@ 3, 20 SAY GOAL_NUM
@ 6, 0  SAY "GOAL DESCRIPTION"
@ 6, 18 GET GOAL_DESCR
@ 7,0 TO 7,79 DOUBLE
@ 8, 0  SAY "TEST PRIORITY"
@ 8, 15 GET TEST_PRI PICTURE "99" RANGE 0,4
@ 8, 28 SAY "AGGREGATE"
@ 8, 34 GET AGGREGATE PICTURE "999" RANGE 0,99
@ 8, 47 SAY "TEST CLASS"
@ 8, 59 GET TEST_CLASS PICTURE "99" RANGE 0,4
@ 10, 0 SAY "ENTER ITERATION FOR READDING LINKS"
@ 10, 36 GET Miterchg PICTURE "99" RANGE 1,99
@ 12, 0 SAY "ITERATION ADDED"
@ 12, 17 GET ITER_ADDED PICTURE "999" RANGE 1,99
@ 14, 0 SAY "ITERATION UPDATED"
@ 15, 0 SAY "Enter current iteration"
@ 14, 19 GET UPDATED PICTURE "999" RANGE 1,99
@ 14, 32 SAY "ITERATION DELETED"
@ 14, 51 SAY "Deleted field auto"
@ 15, 52 SAY "reset to 0"
@ 17, 0 SAY "HISTORY"
@ 17, 9 GET G_HISTORY
@ 17, 17 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 18, 0 SAY "Enter reason for undelete and any link changes"
@ 21, 2 SAY "Press "+CHR(24)+" to make corrections,"
*@ 22, 2 SAY "Y to continue,"
*@ 23, 2 SAY "or N to exit";
* GET Adding PICTURE "!"
@ 0, 0 TO 2, 79 DOUBLE
@ 20, 0 TO 24, 79 DOUBLE

```

```
* Program Name   : DEL_TG.PRG
* Author        : Ned Davis
* Date          : 11 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Accepts user input to mark a test goal as
*                  deleted for a given iteration. Also marks
*                  all associated links as deleted for the
*                  same iteration. Called by UP_GOALS.PRG.
*****
```

**** initialize variables ****

```
g_num = 0
iter = 0
ans = 0
del_it = .F.
```

DO WHILE ans # 2

```
  g_num = 0
  iter = 0
  ans = 0
  del_it = .F.
  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 14,70 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,30 SAY "GOAL DELETION"
  SET COLOR TO W+/BG
  @ 7,20 SAY "Enter the goal number to delete: " GET g_num;
    PICTURE "99999" RANGE 1,9999
  READ
  @ 9,20 SAY "Enter the iteration in which deleted: " GET iter;
    PICTURE "999" RANGE 1,99
  READ
  @ 11,20 SAY "Press 1 to cancel, 2 to exit, "
  @ 12,20 SAY " and return to continue " GET ans PICTURE "99";
    RANGE 0,2
  READ
  SET COLOR TO GR+/BG
```

CLEAR

IF ans # 1

```
  IF ans # 2 .OR. (ans = 2 .AND. g_num # 0)
    USE goals INDEX goal_num, delet_tg
    SEEK g_num
    IF FOUND()
      DISPLAY OFF
      SET COLOR TO W+/BG
      @ 5,5 SAY "Do you want to delete this goal(Y or N)?";
```

```

GET del_it PICTURE "Y"
READ
SET COLOR TO GR+/BG

IF del_it
  REPLACE deleted WITH iter
  USE ro_link INDEX gnumlink, delet_lk
  SEEK g_num
  IF FOUND()
    REPLACE deleted WITH iter WHILE goal_num = g_num
    @ 7,5 SAY "Goal and links deleted"
    WAIT
  ELSE
    @ 7,5 SAY "Goal has no links.  Goal deleted."
    WAIT
  ENDIF
ELSE
  @ 7,5 SAY "Goal NOT deleted"
  WAIT
ENDIF
ELSE
  @ 5,20
  ? "No goal number found for ",g_num
  WAIT
ENDIF
ENDIF
ELSE
  @ 5,0 SAY "Deletion cancelled"
  WAIT
ENDIF
CLOSE DATABASES
ENDDO

CLEAR
RETURN

```

```

*****
* Program Name   : ADD_LINK.PRG
* Author        : Ned Davis
* Date          : 1 Dec 90
* Revised       :
* Language      : dBase III+
* Description    : Builds the RO_LINK.DBF database file.
*                  Manually loaded by the operator. Called
*                  by UP_GOALS.PRG and by UP_PSDL.PRG. Only
*                  links records that exist and are not deleted.
*****

```

```

**** initialize variables ****

```

```

add_link = " "
legal = .F.

```

```

**** select entry screen ****

```

```

SET FORMAT TO addlkscn

```

```

**** start loop for adding links ****

```

```

DO WHILE Add_link # "X"

```

```

    **** initialize memory variables ****

```

```

    Mgoal_num = 0
    Mop_number = 0
    Mpsdl_part = "
    Ml_iteradd = 0
    Mdeleted = 0

```

```

    **** read in values using addlkscn.fmt ****

```

```

    READ
    CLOSE FORMAT

```

```

    IF add_link # "C"

```

```

        IF add_link # "X" .OR. (add_link = "X" .AND. Mgoal_num # 0)

```

```

            USE goals INDEX goal_num

```

```

            GO TOP

```

```

            SEEK Mgoal_num

```

```

            IF FOUND()

```

```

                IF deleted = 0

```

```

                    legal = .T.

```

```

                ELSE

```

```

                    legal = .F.

```

```

                    CLEAR

```

```

                    @ 5,5

```

```

                    ? "Goal number ",Mgoal_num," is deleted."

```

```

                    WAIT

```

```

                ENDIF

```

```

            ELSE

```

```

                legal = .F.

```

```

                CLEAR

```

```

        @ 5,5
        ? "Goal number ",Mgoal_num," not found."
        WAIT
    ENDIF
    IF legal
        USE opstream INDEX op_num
        GO TOP
        SEEK Mop_number
        IF FOUND() .and. legal
            IF deleted = 0
                legal = .T.
            ELSE
                legal = .F.
            CLEAR
            @ 5,5
            ? "Op/Data stream number ",Mop_number," is deleted."
            WAIT
        ENDIF
    ELSE
        legal = .F.
        CLEAR
        @ 5,5
        ? "Op/Data stream number ",Mop_number," not found."
        WAIT
    ENDIF
ENDIF
IF legal
    USL ro_link INDEX gnumlink, onumlink, partlink, l_iterad, i
delet_1k
    APPEND BLANK

    **** store mem vars on new record ****
    REPLACE goal_num WITH Mgoal_num, ;
            op_number WITH Mop_number, ;
            psdl_part WITH UPPER(Mpsdl_part), ;
            l_iteradd WITH Ml_iteradd, ;
            deleted WITH Mdeleted

    CLEAR
    @ 5,5
    ? "Link added for test goal ",Mgoal_num," and Op/DS i
    ",Mop_number,","
    WAIT

    ELSE
        CLEAR
        @ 5,5
        ? "No link added."
        WAIT
    ENDIF
ENDIF
ENDIF
ENDIF

```

CLOSE DATABASES
SET FORMAT TO addlksen
ENDDO

CLOSE DATABASES
CLOSE FORMAT
RETURN

```
* Program Name   : ADDLKSCN.FMT
* Author        : Ned Davis
* Date          : 5 Sep 90
* Revised       :
* Language      : dBase III+
* Description    : Format file for entering new links
*                  between test goals and PSDL operators/
*                  data streams and listing the PSDL grammar
*                  portion where implementation occurs.
*                  Called by ADD_LINK.PRG.
```

```
@ 1, 27 SAY "LINK DATABASE DATA ENTRY"
@ 3, 0 SAY "GOAL NUMBER"
@ 3, 14 GET MGOAL_NUM PICTURE "999999"
@ 3, 25 SAY "OPERATOR/DATA STREAM NUMBER"
@ 3, 54 GET MOP_NUMBER PICTURE "999999"
@ 3, 65 SAY "DELETED"
@ 3, 75 GET MDELETED PICTURE "99"
@ 5, 0 SAY "PSDL PART"
@ 5, 11 GET MPSDL_PART PICTURE "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
@ 7, 0 SAY "ITERATION ADDED"
@ 7, 17 GET MI_ITERADD PICTURE "99"
@ 11, 2 SAY "List the particular PSDL part by its"
@ 12, 2 SAY " grammatical title for a detailed location"
@ 13, 2 SAY "of a goal's implementation."
@ 19, 2 SAY "Press "+CHR(24)+" to make corrections,"
@ 20, 2 SAY "Return to continue, C to cancel,"
@ 21, 2 SAY "or X to exit";
    GET Add_link PICTURE "!"
@ 0, 0 TO 2, 79 DOUBLE
@ 10, 0 TO 14, 79 DOUBLE
@ 18, 0 TO 22, 79 DOUBLE
```

```
* Program Name   : DEL_LINK
* Author        : Ned Davis
* Date          : 11 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Removes link fm. RO_LINK.DBF database file.
*                  Manually loaded by the operator. Called
*                  by UP_GOALS.PRG and by UP_PSDL.PRG.
*****
```

del_link = 0

**** open database ****

USE ro_link INDEX gnumlink, onumlink, partlink, l_iterad,
delet_lk

**** select entry screen ****

SET FORMAT TO dellkscn

**** start loop for adding links ****

DO WHILE del_link # 2

GO TOP

**** initialize memory variables ****

del_link = 0

Mgoal_num = 0

Mop_number = 0

Mpsdl_part = "

Mdeleted = 0

del_it = .F.

**** read in values using dellkscn.fmt ****

READ

IF del_link # 1

IF del_link # 2 .OR. (del_link = 2 .AND. Mgoal_num # 0)

CLEAR

SEEK Mgoal_num

IF FOUND()

DO WHILE goal_num = Mgoal_num .AND. .NOT. del_it .AND. .NOT. EOF()

IF op_number = Mop_number .AND. psdl_part = Mpsdl_part

REPLACE deleted WITH Mdeleted

del_it = .T.

? "Link deleted"

WAIT

ELSE

SKIP

ENDIF

ENDDO

```

ENDIF
IF .NOT. del_it      && search done no link found
? "Link not found for goal no. ",Mgoal_num
?? " op no. ",Mop_number
? " part ",Mpsdl_part
WAIT
ENDIF
ENDIF
ENDIF
ENDDO

CLOSE DATABASES
CLOSE FORMAT
RETURN

```

```
* Program Name : DELLKSCN.FMT
* Author       : Ned Davis
* Date        : 11 Oct. 90
* Revised     :
* Language    : dBase III+
* Description  : Format file for removing links from
                between test goals and PSDL operators/
                data streams and listing the PSDL grammar
                portion where implementation occurs.
                Called by DEL_LINK.PRG.
```

```
@ 1, 32 SAY "LINK DELETION"
@ 3, 0 SAY "GOAL NUMBER OF LINK"
@ 3, 21 GET Mgoal_num PICTURE "99999" RANGE 1,9999
@ 3, 28 SAY "OPERATOR/DATA STREAM NUMBER"
@ 3, 57 GET MOP_NUMBER PICTURE "99999" RANGE 1,9999
@ 5, 0 SAY "ITERATION DELETED"
@ 5, 19 GET MDELETED PICTURE "999" RANGE 1,99
@ 7, 0 SAY "PSDL PART"
@ 7, 11 GET MPSDL_PART PICTURE "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
@ 11, 2 SAY "List the particular PSDL part by its"
@ 12, 2 SAY " grammatical title for a detailed location"
@ 13, 2 SAY "of a goal's implementation."
@ 19, 2 SAY "Press "+CHR(24)+" to make corrections,"
@ 20, 2 SAY "Return to continue, 1 to cancel,"
@ 21, 2 SAY "or 2 to exit";
GET del_link PICTURE "99" RANGE 0,2
@ 0, 0 TO 2, 79 DOUBLE
@ 10, 0 TO 14, 79 DOUBLE
@ 18, 0 TO 22, 79 DOUBLE
```

```

*****
* Program Name: UP_PSDL.PRG
* Author      : Ned Davis
* Date       : 31 Aug 90
* Revised    :
* Language   : dBase III+
* Description : Provides the menu screen for allowing update
*               of the PSDL Operator/Data Stream database file
*               and allows the user to move to other system
*               actions in the TGTS for requirements-based
*               testing in the CAPS.
*****

```

```

**** set environment ****
choice = 0

```

```

**** display PSDL op/data strm update menu ****
DO WHILE choice # 0
  choice = 0
  SET COLOR TO R/BG,GP+/R,G
  CLEAR
  @ 3,10 TO 20,70 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,23 SAY "PSDL OPERATOR/DATA STREAM UPDATE MENU"
  @ 7,20 SAY "Select your choice by number: "
  @ 9,20 SAY "1. Add a new operator/data stream"
  @ 10,20 SAY "2. Change an existing operator/data stream"
  @ 11,20 SAY "3. Delete an existing operator/data stream"
  @ 12,20 SAY "4. Link operator/data stream to test goal"
  @ 13,20 SAY "5. Unlink operator/data stream to test goal"
  @ 14,20 SAY "6. Return to Main Menu"
  SET COLOR TO W+/BG
  @ 17,15 SAY "Select your option: " GET choice PICTURE "99";
  RANGE 0,6
  READ

```

```

**** perform user's request ****
DO CASE
  CASE choice = 1
    DO ADD_PSDL
  CASE choice = 2
    DO CHNGPSDL
  CASE choice = 3
    DO DEL_PSDL
  CASE choice = 4
    DO ADD_LINK
  CASE choice = 5
    DO DEL_LINK
ENDCASE

```

```

ENDDO

```

CLEAR
choice = 0
RETURN TO MASTER

```

*****
*   Program Name   :   ADD_PSDL.PRG
*   Author        :   Ned Davis
*   Date          :   1 Sep 90
*   Revised       :
*   Language      :   Dbase III+
*   Description    :   Allows the addition of one or more new
*                       operators/data strms to the opstream database.
*****

```

```

**** open database ****
USE opstream

```

```

**** find largest current op_number ****
GO BOTTOM
Mop_num = op_number      && assumes largest num is last

```

```

**** select entry screen ****
SET FORMAT TO addopsen

```

```

**** set index files ****
SET INDEX TO op_num, op_name, o_update, o_iterad, delet_op;

```

```

**** start loop for adding op/strm ****
Adding = " "      && memvar for accept, cancel, exit entry scrn
DO WHILE Adding # "X"

```

```

    ****initialize memory variables****
    Mop_name = " "
    Moperator = .T.
    Mupdated = 0
    Mo_iteradd = 0
    Mdeleted = 0
    Adding = " "
    Mop_num = Mop_num + 1

```

```

APPEND BLANK
*--read in values using addopsen.fmt
READ

```

```

IF adding # "C"
    IF adding # "X" .OR. (adding = "X" .AND. Mo_iteradd # 0)

```

```

        *--store memory vars on new record
        REPLACE op_number WITH Mop_num, ;
                op_name WITH Mop_name, ;
                operator WITH Moperator, ;
                o_iteradd WITH Mo_iteradd
        REPLACE updated WITH Mupdated, ;
                deleted WITH Mdeleted

```

```

    ELSE

```

```
        DELETE
        PACK
    ENDIF
ELSE
    DELETE
    PACK
    Mop_num = Mop_num - 1
ENDIF
ENDDO
CLOSE FORMAT
CLOSE DATABASES
RETURN
```

```
* Program Name : ADDOPSCN.FMT
* Author       : Ned Davis
* Date        : 31 Aug 90
* Revised     :
* Language    : dBase III+
* Description  : Format file for entering new PSDL operator
                or data stream to the opstream data base.
                Called by ADD_PSDL.PRG. Does not allow
                update or deletion to be flagged.
```

```
@ 1, 26 SAY "OPSTREAM DATABASE DATA ENTRY"
@ 3, 0  SAY "OPERATOR/DATA STREAM NUMBER"
*--auto insert next goal number
@ 3, 30 SAY MOP_NUM
@ 5, 0  SAY "OPERATOR/DATA STREAM NAME"
@ 5, 27 GET MOP_NAME PICTURE
"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
@ 7, 0  SAY "OPERATOR"
@ 7, 10 GET MOPERATOR PICTURE "L"
@ 7, 23 SAY "ITERATION ADDED"
@ 7, 40 GET MO_ITERADD PICTURE "999" RANGE 1,99
@ 8, 0  SAY "Enter T if operator or F if data stream"
@ 10, 0 SAY "HISTORY"
@ 10, 9 GET OPSTREAM->O_HISTORY
@ 10, 15 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 12, 2 SAY "Press "+CHR(24)+" to make corrections,"
@ 12,37 SAY "Operator field - Enter 'T' or 'Y'"
@ 13, 2 SAY "return to continue, C to cancel,"
@ 13,37 SAY "if operator, else 'P' or 'N'"
@ 14, 2 SAY "or X to exit.";
        GET Adding PICTURE "!"
@ 14,37 SAY "if data stream"
@ 0, 0  TO 2, 79 DOUBLE
@ 11, 0 TO 15, 79 DOUBLE
@ 12,35 TO 14, 35
```

```

*****
* Program Name   : CHNGPSDL.PRG
* Author        : Ned Davis
* Date          : 1 Dec 90
* Revised       :
* Language      : dBase III+
* Description    : Allows user to update 1 record at a time
*                  of the OPSTREAM.DBF database. All fields may
*                  be modified except op_num field.
*                  A deleted goal may be undeleted and its
*                  related link records may be block undeleted
*                  or selectively undeleted. op/data strms are not
*                  deleted by this program. Called by
*                  UP_PSDL.PRG.
*****

```

```

**** set environment ****
SET MEMOWIDTH TO 20    && lets display output fit on 1 line

```

```

**** initialize memory variables ****
Adding = "Y"          && loop flag
Change_it = "Y"       && choice variable
Mop_num = 0           && dummy var to get goal number
choice_a = 2

```

```

**** start loop for changing operator/data streams ****
DO WHILE Adding # "N"
  CLOSE DATABASES

```

```

**** select database ****
USE opstream INDEX i
  op_num, op_name, o_update, o_iterad, delet_op

```

```

**** display input screen ****
SET FORMAT TO opstmchg
READ

```

```

**** det'm whether to change record ****
IF Adding = "N"      && decided not to change record
  CLOSE FORMAT
  LOOP
ENDIF

```

```

**** prep screen ****
CLOSE FORMAT
CLEAR

```

```

**** look for record ****
SEEK Mop_num
IF FOUND()
  *--show it to the user for confirmation

```

```

DISPLAY op_number,op_name,o_history OFF

*--get confirmation
Change_it = "Y"    && set default
@ 22,0 SAY "Do you want to change this record? (Y or N)"
@ 22,45 GET Change_it PICTURE "Y"
READ

*--act on confirmation
IF Change_it = "N"    && don't change this record
  LOOP
ELSE    && change record
  CLEAR
  IF deleted = 0    && record is in use
    SET FORMAT TO chgopscn
    READ
    WAIT "Op/Data Stream updated, press any key to continue"
    CLOSE FORMAT
  ELSE    && record is deleted
    choice_a = 2,    && set default choice
    @ 0,0 SAY "Operator is currently deleted"
    @ 3,0 SAY "Options:"
    @ 5,5 SAY "1. Undelete and make appropriate changes"
    @ 6,5 SAY "2. Disregard op/data stream change"
    @ 8,5 SAY "Select your choice: "
    @ 8,25 GET choice_a PICTURE "99" RANGE 1,2
    READ

    ***** act upon choice *****
    DO CASE
      CASE choice_a = 1    && undelete
        DO CHGPHLP
      CASE choice_a = 2    && disregard
        LOOP
    ENDCASE
  ENDIF
ENDIF
ELSE
  *--if not found, warn user
  @ 22,0
  ? "Can't find operator/data stream: ",Mop_num
  ?? CHR(7)
  WAIT
ENDIF
Mop_num = 0
ENDDO
CLOSE DATABASES
SET MEMOWIDTH TO 30
RETURN

```

* Program Name : OPSTMCHG.PMT
* Author : Ned Davis
* Date : 18 Sep 90
* Revised :
* Language : dBase III+
* Description : Screen for inputting op/data stream number to
* change or exit w/o change. Starts modification
* of OPSTREAM.DBF data base file one record at a
* time. May also modify RO_LINK.DBF if a
* goal previously deleted is undeleted.
* Called by CHNGPSDL.PRG.

@ 5, 5 SAY "Enter operator/data stream number (or 0) of OP/DS to
change:"
@ 5,68 GET Mop_num PICTURE "999999"
@ 7, 5 SAY "Enter N to exit w/o change, Y to continue"
@ 7,58 GET Adding PICTURE "Y"

```

*****
*   Program Name:  CHGOPSCN.FMT
*   Author       :  Ned Davis
*   Date        :  18 Sep 90
*   Revised     :
*   Language    :  dBase III+
*   Description  :  Format file for changing op/data stream in
*                   the OPSTREAM.DBF database file. Called by
*                   CHNGPSDL.PRG. Does not allow deletion or
*                   undeletion. Used exclusively for records
*                   not currently deleted.
*****

@ 1, 20 SAY "OPERATOR/DATA STREAM DATABASE DATA CHANGE"
@ 3, 0 SAY "OPERATOR/DATA STREAM NUMBER"
@ 3, 31 SAY OP_NUMBER
@ 6, 0 SAY "OPERATOR NAME"
@ 6, 15 GET OP_NAME
      PICTURE "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
@ 8, 0 SAY "OPERATOR"
@ 8, 10 GET OPERATOR PICTURE "L"
@ 9, 0 SAY "Enter T if operator and F if data stream"
@ 10, 0 SAY "ITERATION ADDED"
@ 10, 17 GET O_ITERADD PICTURE "999" RANGE 1,99
@ 12, 0 SAY "ITERATION UPDATED"
@ 12, 19 GET UPDATED PICTURE "999" RANGE 0,99
@ 12, 32 SAY "ITERATION DELETED"
@ 12, 51 SAY DELETED
@ 15, 0 SAY "HISTORY"
@ 15, 9 GET O_HISTORY
@ 15, 17 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 19, 2 SAY "Press "+CHR(24)+" to make corrections,"
*@ 20, 2 SAY "Return to continue, C to Cancel,"
*@ 21, 2 SAY "or X to exit";
* GET Adding PICTURE "!"
@ 0, 0 TO 2, 79 DOUBLE
@ 18, 0 TO 22, 79 DOUBLE

```

```

*****
* Program Name   : CHGPHLP.PRG
* Author        : Ned Davis
* Date          : 1 Dec 90
* Revised       :
* Language      : dBase III+
* Description    : Allows user to update record of OPSTREAM.DBF
                  and links of RO_LINK.DBF database. All fields may
                  be modified except op_num field.
                  A deleted goal is undeleted and its
                  related link records may be block undeleted
                  or selectively undeleted. op/data strms are not
                  deleted by this program. Called by
                  CHNGPSDL.PRG. Uses global vars.
*****

```

```

**** initialize memory variables ****

```

```

Miterchg = 0

```

```

ch_link = 1

```

```

**** act upon choice ****

```

```

SET FORMAT TO chopscnl

```

```

READ

```

```

REPLACE deleted WITH 0

```

```

CLOSE FORMAT

```

```

CLEAR

```

```

@ 5,5

```

```

WAIT "Op/data stream updated, press any key to continue"

```

```

CLOSE FORMAT

```

```

*--undelete link records. Change database.

```

```

USE ro_link INDEX onumlink

```

```

SEEK Mop_num

```

```

IF FOUND()      && link record(s) found

```

```

    CLEAR

```

```

    ch_link = 1      && set default choice

```

```

    @ 5,30 SAY "UNDELETE LINK OPTIONS"

```

```

    @ 7,15 SAY "Options:"

```

```

    @ 9,15 SAY "1. Batch undelete all links to PSDL part"

```

```

    @ 10,15 SAY "2. Sequence through links for selective undelete"

```

```

    @ 12,15 SAY "Enter option 1 or 2:"

```

```

    @ 12,37 GET ch_link PICTURE "99" RANGE 1,2

```

```

    READ

```

```

DO CASE

```

```

    CASE ch_link = 1

```

```

        DO WHILE op_number = Mop_num

```

```

            REPLACE deleted WITH 0

```

```

            REPLACE l_iteradd WITH Miterchg

```

```

            SKIP

```

```

        ENDDO

```

```

CASE ch_link = 2
  DO WHILE op_number = Mop_num
    CLEAR
    DISPLAY goal_num,op_number,psdl_part,deleted,l_iteradd OFF
    @ 5,5 SAY "Enter Y to undelete, N to skip"
    @ 5,37 GET Adding PICTURE "Y"
    READ
    IF Adding = "Y"
      REPLACE deleted WITH 0
      REPLACE l_iteradd WITH Miterchg
      SKIP
    ELSE
      SKIP
    ENDIF
  ENDDO
ENDCASE

**** update index file ****
SET INDEX TO delet_lk
REINDEX
CLOSE INDEX
CLEAR
@ 5,5
WAIT "Links updated, press any key to continue"
ELSE
  && no link records found
  CLEAR
  @ 5,5 SAY "No link record found."
  WAIT
ENDIF
CLOSE DATABASES
RETURN

```

```
* Program Name:  CHOPSCN1.FMT
* Author       :  Ned Davis
* Date        :  18 Sep 90
* Revised     :
* Language    :  dBase III+
* Description :  Format file for changing op/data stream in
*                the OPSTREAM.DBF database file. Called by
*                CHGPHLP.PRG. Used explicitly with deleted
*                opstream records that are to be undeleted. Aids
*                in ensuring that info that must change with
*                an undelete gets changed.
```

```
@ 1, 15 SAY "OPERATOR/DATA STREAM DATABASE DATA CHANGE FOR UNDELETE"
@ 3, 0  SAY "OPERATOR/DATA STREAM NUMBER"
@ 3, 30 SAY OP_NUMBER
@ 6, 0  SAY "OPERATOR NAME"
@ 6, 16 GET OP_NAME
      PICTURE "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
@ 7,0 TO 7,79 DOUBLE
@ 8, 0  SAY "ENTER ITERATION FOR READDING LINKS"
@ 8, 36 GET miterchg PICTURE "999" RANGE 1,99
@ 10, 0 SAY "ITERATION ADDED"
@ 10, 17 GET O_ITERADD PICTURE "999" RANGE 1,99
@ 10, 25 SAY "IS AN OPERATOR"
@ 10, 41 GET OPERATOR PICTURE "L"
@ 11, 25 SAY "Enter T if operator or F if data stream"
@ 12, 0  SAY "ITERATION UPDATED"
@ 13, 0  SAY "Enter current iteration"
@ 12, 19 GET UPDATED PICTURE "999" RANGE 1,99
@ 12, 32 SAY "ITERATION DELETED"
@ 12, 51 SAY "Deleted field auto"
@ 13, 51 SAY "reset to 0"
@ 15, 0  SAY "HISTORY"
@ 15, 9  GET O_HISTORY
@ 15, 17 SAY "Move cursor to memo field, type Ctrl-PgDn"
@ 16, 0  SAY "Enter reason for undelete and any link changes"
@ 19, 2  SAY "Press "+CHR(24)+" to make corrections,"
*@ 20, 2  SAY "Return to continue, C to Cancel,"
*@ 21, 2  SAY "or X to exit";
* GET Adding PICTURE "!"
@ 0, 0 TO 2, 79 DOUBLE
@ 18, 0 TO 22, 79 DOUBLE
```

```
* Program Name   : DEL_PSDL.PRG
* Author        : Ned Davis
* Date          : 11 Oct 90
* Revised       :
* Language      : dBase III+
* Description    : Accepts user input to mark an op/data strm as
*                  deleted for a given iteration. Also marks
*                  all associated links as deleted for the
*                  same iteration. Called by UP_PSDL.PRG.
*****
```

**** initialize variables ****

```
o_num = 0
iter = 0
ans = 0
del_it = .F.
```

DO WHILE ans # 2

```
o_num = 0
iter = 0
ans = 0
del_it = .F.
SET COLOR TO R/BG,GR+/R,G
CLEAR
@ 3,10 TO 14,70 DOUBLE
SET COLOR TO GR+/BG
@ 5,25 SAY "OPERATOR/DATA STREAM DELETION"
SET COLOR TO W+/BG
@ 7,20 SAY "Enter the op/data stream number to delete: " GET o_num;
    PICTURE "99999" RANGE 1,9999
READ
@ 9,20 SAY "Enter the iteration in which deleted: " GET iter;
    PICTURE "999" RANGE 1,99
READ
@ 11,20 SAY "Press 1 to cancel, 2 to exit, "
@ 12,20 SAY " and return to continue " GET ans PICTURE "99";
    RANGE 0,2
```

```
READ
SET COLOR TO GR+/BG
```

CLEAR

IF ans # 1

IF ans # 2 .OR. (ans = 2 .AND. o_num # 0)

USE opstream INDEX op_num, delet_op

SEEK o_num

IF FOUND()

DISPLAY OFF

SET COLOR TO W+/BG

@ 5,5 SAY "Do you want to delete this op/data stream(Y or N)?";

```

GET del_it PICTURE "Y"
READ
SET COLOR TO GR+/BG

IF del_it
  REPLACE deleted WITH iter
  USE ro_link INDEX onumlink, delet_lk
  SEEK o_num
  IF FOUND()
    REPLACE deleted WITH iter WHILE op_number = o_num
    @ 7,5 SAY "Operator/Data Stream and links deleted"
    WAIT
  ELSE
    @ 7,5 SAY "Operator/Data stream has no links. i
Operator/Data stream deleted."
    WAIT
  ENDIF
ELSE
  @ 7,5 SAY "Operator/Data Stream NOT deleted"
  WAIT
ENDIF
ELSE
  @ 5,20
  ? "No Operator/Data Stream number found for ",o_num
  WAIT
ENDIF
ENDIF
ELSE
  @ 5,0 SAY "Deletion cancelled"
  WAIT
ENDIF
CLOSE DATABASES
ENDDO

CLEAR
RETURN

```

```

*****
* Program Name : ITERINFO.PRG
* Author      : Ned Davis
* Date       : 2 Dec 90
* Revised    :
* Language   : Dbase III+
* Description : Displays a menu to add, change, delete or
*               change records on each iteration of proto_
*               type development. Also will give you the
*               highest iteration number used to date.
*               Called by TESTGOAL.PRG
*****

```

```

**** initialize variables ****
ans = 0

```

```

DO WHILE ans # 7
  ans = 0
  SET COLOR TO R/BG,GR+/R,G
  CLEAR
  @ 3,10 TO 20,70 DOUBLE
  SET COLOR TO GR+/BG
  @ 5,25 SAY "PROTOTYPE ITERATION INFORMATION"
  @ 7,20 SAY "Select your choice by number: "
  @ 9,20 SAY "1. Add a new iteration"
  @ 10,20 SAY "2. Modify an existing iteration"
  @ 11,20 SAY "3. Delete an existing iteration"
  @ 12,20 SAY "4. List all iteration information"
  @ 13,20 SAY "5. List information for iteration x"
  @ 14,20 SAY "6. List most recent iteration no."
  @ 15,20 SAY "7. Return to Main Menu"
  SET COLOR TO W+/BG
  @ 17,20 SAY "Select your option: " GET ans PICTURE "99";
  RANGE 0,7
  READ
  SET COLOR TO GR+/BG

```

```

**** perform request ****

```

```

CLEAR
DO CASE
  CASE ans = 1
    DO I_ADD
  CASE ans = 2
    DO I_CHNG
  CASE ans = 3
    DO I_DEL
  CASE ans = 4
    DO ALL_IOUT
  CASE ans = 5
    DO ITER_OUT
  CASE ans = 6

```

DO LASTITER
ENDCASE

ENDDO
CLEAR
RETURN TO MASTER

```

*****
* Program Name : I_ADD.PRG
* Author       : Ned Davis
* Date        : 2 Dec 90
* Revised     :
* Language    : Dbase III+
* Description  : Adds a record to the ITERATNS.DBF database.
*               Auto increments iteration number.
*               Called by ITERINFO.PRG
*****

```

```

**** initialize variables ****

```

```

do_it = 2
Miter_num = 0

```

```

**** initialize database ****
USE iteratns INDEX it_num

```

```

**** find largest current iter no. ****
GO TOP

```

```

*wait

```

```

DO WHILE .NOT. EOF()

```

```

*wait

```

```

    IF iter_num > Miter_num
        Miter_num = iter_num
    ENDIF
    SKIP
ENDDO

```

```

SET FORMAT TO add_iter
Miter_num = Miter_num + 1
APPEND BLANK
REPLACE iter_num WITH Miter_num
READ
CLOSE FORMAT
IF do_it = 1
    DELETE
    PACK
    CLEAR
    @ 5,5
    ? "Entry cancelled."
    WAIT
ELSE
    CLEAR
    ? "Record added"
    WAIT
ENDIF

```

CLEAR
CLOSE DATABASES
RETURN

* Program Name : ADD_ITER.FMT
* Author : Ned Davis
* Date : 2 Dec 90
* Revised :
* Language : dBase III+
* Description : Screen format file for adding new iteration
* information to the ITERATNS.DBF database
* file. Called by ITERINFO.PRG.
*

@ 1,25 SAY "ADD AN ITERATION HISTORY"
@ 3,25 SAY "NEW ITERATION NUMBER"
@ 3,47 SAY iter_num
@ 5,25 SAY "NEW ITERATION HISTORY"
@ 5,48 GET i_history
@ 6,20 SAY "Cntl+PgDn to enter, Cntl+PgUp to exit"
@ 8,15 SAY "Enter 1 to cancel, 2 to accept entry"
@ 8,53 GET do_it PICTURE "99" RANGE 1,2
@ 7, 1 TO 7,69
@ 0, 0 TO 9,70 DOUBLE

```

*****
* Program Name : I_CHNG.PRG
* Author       : Ned Davis
* Date        : 2 Dec 90
* Revised     :
* Language    : Dbase III+
* Description  : Changes existing record in ITERATNS.DBF.
*              Called by ITERINFO.PRG.
*****

```

```

**** initialize database ****
USE iteratns INDEX it_num

```

```

**** initialize variables ****
Miter_num = 0
do_it = 2

```

```

CLEAR
@ 5,5 SAY "Enter the iteration to change: "GET Miter_num;
      PICTURE "999" RANGE 1,99
READ
SEEK Miter_num
IF FOUND()
    SET FORMAT TO chg_iter
    READ
    CLOSE FORMAT
ELSE
    ? "Did not find iteration: ",Miter_num
    WAIT
ENDIF
CLEAR
CLOSE DATABASES
RETURN

```

* Program Name : CHG_ITER.FMT
* Author : Ned Davis
* Date : 2 Dec 90
* Revised :
* Language : dBase III+
* Description : Screen format file for changing iteration
* information in the ITERATNS.DBF database
* file. Called by ITERINFO.PRG.
*

@ 1,20 SAY "CHANGE AN ITERATION HISTORY"
@ 3,20 SAY "ITERATION NUMBER"
@ 3,42 GET iter_num PICTURE "999" RANGE 1,99
@ 5,20 SAY "ITERATION HISTORY"
@ 5,43 GET i_history
@ 6,20 SAY "Cntl-PgDn to enter, Cntl-PgUp to exit"
@ 8,15 SAY "Enter 1 to continue."
@ 8,53 GET de_it PICTURE "99" RANGE 1,1
@ 7, 1 TO 7,69
@ 0, 0 TO 9,70 DOUBLE

```
* Program Name : I_DEL.PRG
* Author       : Ned Davis
* Date        : 2 Dec 90
* Revised     :
* Language    : Dbase III+
* Description  : Allows record delete for ITERATNS.DBF.
*              Called by ITERINFO.PRG.
```

**** initialize database ****

USE iteratns INDEX it_num

**** initialize variables ****

Miter_num = 0

del_it = .F.

do_it = 2

@ 5,5 SAY "Enter the iteration to delete: "GET Miter_num;
PICTURE "999" RANGE 1,99

READ

SEEK Miter_num

IF FOUND()

 CLEAR

 SET MEMOWIDTH TO 60

 DISPLAY ITER_NUM, I_HISTORY OFF

 SET MEMOWIDTH TO 30

 @ 23,0 SAY "Do you want to delete this iteration(Y or N)?"

 @ 23,48 GET del_it PICTURE "Y"

 READ

 IF del_it

 DELETE

 PACK

 CLEAR

 ? "Iteration ",mITER_NUM," deleted."

 WAIT

 ENDIF

ELSE

 ? "Did not find iteration: ",Miter_num

 WAIT

ENDIF

CLEAR

CLOSE DATABASES

RETURN

```

*****
* Program Name : ALL_ICUT.PRG
* Author      : Ned Davis
* Date       : 2 Dec 90
* Revised    :
* Language   : Dbase III+
* Description : Displays all ITERATNS.DBF records to screen
*              or to screen and printer. Called by
*              ITERINFO.PRG.
*****

```

```

**** initialize database ****
USE iteratns INDEX it_num

```

```

**** initialize variables ****
Miter_num = 0
output = 1

```

```

CLEAR
@ 5,5 SAY "Enter 1 to output to screen"
@ 6,5 SAY " or 2 to output to screen and printer";
      GET output PICTURE "99" RANGE 1,2
READ
SET MEMOWIDTH TO 60
IF output = 1
  DISPLAY ALL ITER_NUM, I_HISTORY OFF
  WAIT
ELSE
  DISPLAY ALL ITER_NUM, I_HISTORY OFF TO PRINT
  WAIT
ENDIF
SET MEMOWIDTH TO 30
CLEAR
CLOSE DATABASES
RETURN

```

```
* Program Name : ITER_OUT.PRG
* Author      : Ned Davis
* Date       : 2 Dec 90
* Revised    :
* Language   : dBase III+
* Description : Displays input fields for outputting an
                iteration information record for the specified
                iteration to the screen or screen and printer.
*             Called by ITERINFO.PRG.
```

**** initialize database ****

USE iteratns INDEX it_num

**** initialize variable ****

Miter_num = 0

output = 1

@ 5,5 SAY "Enter the iteration to output" GET Miter_num;
 PICTURE "999" RANGE 1,99

READ

SEEK Miter_num

IF FOUND()

SET MEMOWIDTH TO 60

@ 7,5 SAY "Enter 1 to output to screen, 2 to output"

@ 8,5 SAY "to screen and printer: " GET output;

PICTURE "99" RANGE 1,2

READ

IF output = 1

DISPLAY ITER_NUM, I_HISTORY OFF

WAIT

ELSE

DISPLAY ITER_NUM, I_HISTORY OFF TO PRINT

WAIT

ENDIF

ELSE

? "Did not find iteration number ",Miter_num

WAIT

ENDIF

SET MEMOWIDTH TO 30

RETURN

```

*****
* Program Name : LASTITER.PRG
* Author       : Ned Davis
* Date        : 2 Dec 90
* Revised     :
* Language    : Dbase III+
* Description  : Displays the highest iteration number
*               used to date. Called by TESTGOAL.PRG.
*****

```

```

**** initialize variables ****
Miter_num = 0

```

```

**** initialize database ****
USE iteratns INDEX it_num

```

```

DO WHILE .NOT. EOF()
  IF iter_num > Miter_num
    Miter_num = iter_num
  ENDIF
  SKIP
ENDDO
@ 5,10 TO 7,70 DOUBLE
@ 6,20 SAY "The most recent iteration is: "
@ 6,52 SAY Miter_num
@ 9,0
WAIT
CLEAR
CLOSE DATABASES
RETURN

```

```

*****
*   Program Name   :   BYE.PRG
*   Author        :   Ned Davis
*   Date          :   16 Aug 90
*   Revised       :
*   Language      :   Dbase III+
*   Description    :   Displays a closing screen for the TESTGOAL
*                     system.
*****

```

CLOSE ALL

```

****display screen****
SET COLOR TO GR/B,GR+/R,G
CLEAR
@ 10,26 TO 14,51 DOUBLE
SET COLOR TO BG+/B
@ 12,30 SAY "GOOD BYE FOR NOW"

```

* Program Name : INDEX_IT.PRG
* Author : Ned Davis
* Date : 3 Sep 90
* Revised :
* Language : dBase III+
* Description : This reindexes all the database files
* and updates all index files.

USE GOALS INDEX GOAL_NUM,TEST_PRI,AGGRFGAT,UPDATED, ITER_ADD,
TESTCLAS, DELET_TG
REINDEX
CLOSE DATABASES

USE OPSTREAM INDEX OP_NUM, OP_NAME, O_UPDATE, O_ITERAD, DELET_OP
REINDEX
CLOSE DATABASES

USE RO_LINK INDEX GNUMLINK, ONUMLINK, PARTLINK, L_ITERAD, DELET_LK
REINDEX
CLOSE ALL

REFERENCES

Agresti, William, *New Paradigms for Software Development*. IEEE Computer Society Press, Washington D.C., 1986.

American National Standards Institute/Institute of Electrical and Electronics Engineers Standard 729-1983, *Standard Glossary of Software Engineering Terminology*. 1983.

Beizer, Boris, *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York, 1983.

Bersoff, E., Gregor, B. and Davis, A., *Alternate Lifecycle Models*, BTG Inc., Vienna, Virginia, 1988.

Boehm, B., "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Transactions on Software Engineering*, v.SE-10, no.3, pp.290-302, May 1984.

Boehm, Barry, *Software Risk Management*. IEEE Computer Society Press, Washington, D.C., 1989.

Davis, A., Bersoff, E., and Comer, E., "A Strategy for Comparing Alternate Software Development Lifecycle Models," *IEEE Transactions on Software Engineering*, v.14, no.10, pp.1453-1461, October 1988.

Fountain, Harrison, *Rapid Prototyping: A Survey and Evaluation of Methodologies and Models*. M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1990.

Frankl, P.G., *ASSET Reference Manual, (Preliminary Draft)*, Department of Computer Science, Courant Institute of Computer Science, New York University, New York, June 29, 1987.

Frankl, P.G. and Weyuker, E.J., "Data Flow Testing in the Presence of Unexecutable Paths," *Proceedings of the IEEE Workshop on Software Testing*, Banff, Canada, pp.4-13, July 1986.

Gomaa, H. and Scott, D., "Prototyping as a Tool in the Specification of User Requirements," *The Proceedings of the 5th International Conference on Software Engineering*, pp.333-342, 1981.

Hernandez, Jr., Jose, *Derivation Strategies for Experienced-Based Test Oracles*. M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1989.

Hetzel, B., *The Complete Guide to Software Testing*, 2nd ed., QED Information Sciences, Inc. 1988.

Howden, W.E., "Functional Testing and Design Abstractions," *The Journal of Systems and Software*, Elsevier North Holland, Inc., pp.307-313, 1980.

Howden, W.E., "A Survey of Dynamic Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society, pp. 209-231, 1981.

Howden, W.E., "A Survey of Static Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society, pp. 101-115, 1981.

Kemmerer, R.A. and Eckmann, S.T., "UNISEX: A UNIX-based Symbolic EXecutor for Pascal," *Software - Practice and Experience*, v.1515, pp.439-458, May 1985.

Kraemer, B. Luqi, and Berzins, V., "A Formal Semantics for PSDL," submitted to *IEEE Transactions on Software Engineering*.

Luqi, "Software Evolution Through Rapid Prototyping," *IEEE Computer*, pp.13-25, May 1989.

Luqi and Berzins, V., "Rapidly Prototyping Real-Time Systems," *IEEE Software*, pp.25-36, September 1988.

Luqi, Berzins, V. and Yeh, R., "A Prototyping Language for Real Time Software". *IEEE Transactions on Software Engineering*, pp. 1409-1423, October 1988.

Manna, A. and Waldinger, R., "The Logic of Computer Programming," *IEEE Transactions on Software Engineering*, v. SE-4, pp.199-229, April 1988.

Naval Postgraduate School Technical Report NPS52-87-012, *Execution of Real-Time Prototypes*, by Luqi, 1987.

Naval Postgraduate School Technical Report NPS52-90-030, *TWIRP: Testing Within Iterative Rapid Prototyping*, by T.F. Shimeall, July 1990

Pressman, R.S., *Software Engineering: A Practitioner's Approach*. 2d ed., McGraw-Hill, Inc., New York, New York, 1987.

Taylor, T. and Standish, T.A., "Initial Thoughts on Rapid Prototyping Techniques," *New Paradigms for Software Development*, Washington: IEEE Computer Society Press, pp.38-47, 1986.

"*The American Heritage Dictionary of the English Language*," Houghton Mifflin Co., Boston, Massachusetts, 1978.

Weyuker, E.J., "The Evaluation of Program-based Software Test Data Adequacy Criteria," *Communications of the ACM*, v.31, pp.668-675, June 1988.

White, Laura, *The Development of a Rapid Prototyping Environment*. M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1989.

BIBLIOGRAPHY

Agresti, William, *New Paradigms for Software Development*. IEEE Computer Society Press, Washington, D.C., 1986.

American National Standards Institute/Institute of Electrical and Electronics Engineers Standard 729-1983, *Standard Glossary of Software Engineering Terminology*. 1983.

Barr, A., Cohen, P.R., and Feigenbaum, E.A., *The Handbook of Artificial Intelligence*. v.4, Addison Wesley, Reading, Massachusetts, 1989.

Beizer, Boris, *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, New York, New York, 1984.

Beizer, Boris, *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York, 1983.

Bersoff, E., Gregor, B. and Davis, A., *Alternate Lifecycle Models*, BTG Inc., Vienna, Virginia, 1988.

Boehm, B., "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Transactions on Software Engineering*, v.SE-10, no.3, pp.290-302, May 1984.

Boehm, Barry, *Software Risk Management*. IEEE Computer Society Press, Washington, D.C., 1989.

Davis, A., Bersoff, E., and Comer, E., "A Strategy for Comparing Alternate Software Development Lifecycle Models," *IEEE Transactions on Software Engineering*, v.14, no.10, pp.1453-1461, October 1988.

Fountain, Harrison, *Rapid Prototyping: A Survey and Evaluation of Methodologies and Models*. M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1990.

Frankl, P.G., *ASSET Reference Manual, (Preliminary Draft)*, Department of Computer Science, Courant Institute of Computer Science, New York University, New York, June 29, 1987.

Frankl, P.G. and Weyuker, E.J., "Data Flow Testing in the Presence of Unexecutable Paths," *Proceedings of the IEEE Workshop on Software Testing*, Banff, Canada, pp. 4-13, July 1986.

Gomaa, H. and Scott, D., "Prototyping as a Tool in the Specification of User Requirements," *The Proceedings of the 5th International Conference on Software Engineering*, pp.333-342, 1981.

Hernandez, Jr., Jose, *Derivation Strategies for Experienced-Based Test Oracles*. M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1989.

Hetzl, B., *The Complete Guide to Software Testing*, 2nd ed., QED Information Sciences, Inc. 1988.

Howden, W.E., "Functional Testing and Design Abstractions," *The Journal of Systems and Software*, Elsevier North Holland, Inc., pp.307-313, 1980.

Howden, W.E., "A Survey of Dynamic Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society, pp. 209-231, 1981.

Howden, W.E., "A Survey of Static Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society, pp. 101-115, 1981.

Kemmerer, R.A. and Eckmann, S.T., "UNISEX: A UNIX-based Symbolic EXecutor for Pascal," *Software - Practice and Experience*, v.1515, pp.439-458, May 1985.

Kraemer, E. Luqi, and Berzins, V., "A Formal Semantics for PSDL," submitted to *IEEE Transactions on Software Engineering*.

Luqi, "Software Evolution Through Rapid Prototyping," *IEEE Computer*, pp.13-25, May 1989.

Luqi and Berzins, V., "Rapidly Prototyping Real-Time Systems," *IEEE Software*, pp.25-36, September 1988.

Luqi, Berzins, V. and Yeh, R., "A Prototyping Language for Real Time Software". *IEEE Transactions on Software Engineering*, pp. 1409-1423, October 1988.

Manna, A. and Waldinger, R., "The Logic of Computer Programming," *IEEE Transactions on Software Engineering*, v. SE-4, pp.199-229, April 1988.

Naval Postgraduate School Technical Report NPS52-87-012, *Execution of Real-Time Prototypes*, by Luqi, 1987.

Naval Postgraduate School Technical Report NPS52-90-030, *TWIRP: Testing Within Iterative Rapid Prototyping*, by T.F. Shimeall, July 1990

Pressman, R.S., *Software Engineering: A Practitioner's Approach*. 2d ed., McGraw-Hill, Inc., New York, New York, 1987.

Taylor, T. and Standish, T.A., "Initial Thoughts on Rapid Prototyping Techniques," *New Paradigms for Software Development*, Washington: IEEE Computer Society Press, pp.38-47, 1986.

"The American Heritage Dictionary of the English Language," Houghton Mifflin Co., Boston, Massachusetts, 1978.

Weyuker, E.J., "The Evaluation of Program-based Software Test Data Adequacy Criteria," *Communications of the ACM*, v.31, pp.668-675, June 1988.

White, Laura, *The Development of a Rapid Prototyping Environment*. M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1989.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145
2. Library, Code 52 2
Naval Postgraduate School
Monterey, California 93943-5002
3. Commandant of the Marine Corps 1
Code TE 06
Headquarters, U.S. Marine Corps
Washington, D.C. 20380-0001
4. Professor Timothy J. Shimeall, Code CSSm 6
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5000
5. Professor Luqi, Code CSLq 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5000
6. Captain Patrick D. Barnes, USAF, Code CSBa 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5000
7. Major Edward V. Davis, Jr., USMC 2
109 Winter Quarters Drive
Pocomoke City, Maryland 21851